Rethinking Security of Web-Based System Applications

Martin Georgiev
University of Texas at Austin
mgeorgiev@utexas.edu

Suman Jana
University of Texas at Austin
suman@cs.utexas.edu

Vitaly Shmatikov
University of Texas at Austin
shmat@cs.utexas.edu

ABSTRACT

Many modern desktop and mobile platforms, including Ubuntu, Google Chrome, Windows, and Firefox OS, support so called Web-based system applications that run outside the Web browser and enjoy direct access to native objects such as files, camera, and geolocation. We show that the access-control models of these platforms are (a) incompatible and (b) prone to unintended delegation of native-access rights: when applications request native access for their own code, they unintentionally enable it for untrusted third-party code, too. This enables malicious ads and other third-party content to steal users' OAuth authentication credentials, access camera on their devices, etc.

We then design, implement, and evaluate POWERGATE, a new access-control mechanism for Web-based system applications. It solves two key problems plaguing all existing platforms: security and consistency. First, unlike the existing platforms, POWERGATE correctly protects native objects from unauthorized access. Second, POWERGATE provides uniform access-control semantics across all platforms and is 100% backward compatible. POWERGATE enables application developers to write well-defined native-object access policies with explicit principals such as "application's own local code" and "third-party Web code," is easy to configure, and incurs negligible performance overhead.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls

Keywords

Web Security; Mobile Security; Firefox OS

1 Introduction

Web-based system applications, as defined by the W3C Working Group [47], are Web applications with comparable capabilities to native applications. They are implemented in languages such as HTML5 and JavaScript but operate outside the Web browser sandbox, with direct access to the operating system on the host machine. Many modern operating systems, including Ubuntu, Google Chrome, Windows, and Firefox OS, provide runtime environments to support Web-based system applications, eroding the distinction between desktop, mobile, and Web-based software.

Web-based system applications offer several attractive features. First, they are implemented in platform-independent Web languages and thus are portable, in contrast to native applications. In theory, the developer can write the application once and it will run on many desktop and mobile platforms. Second, they have access to native functionality such as local files, camera, microphone, etc., in contrast to conventional Web applications. Third, after they are installed by the user, they can work offline, also in contrast to conventional Web applications. Fourth, they are easy to maintain and update by changing the Web code hosted on the developer's site, instead of requiring all users to update their local copies. Fifth, even though they use runtime Web-browser libraries, they do not rely on browsers' user interfaces, which can take up valuable display space on mobile devices and make applications less responsive.

Our contributions. We first analyze the security models of the existing platforms for Web-based system applications: Ubuntu, Chrome, Windows, and Firefox OS. These platforms have different access-control semantics for native objects, making it difficult to implement secure, portable applications even when the language and API are platform-independent (e.g., Open Web APIs in HTML5).

Another problem with the existing platforms is that their access-control policies are expressed in terms of system components and thus do not map naturally onto the applications' privilege separation requirements. For example, in Ubuntu and Windows, native-access capabilities are granted not to the application per se but to specific browser instances. Consider a Web-based system application created from a conventional Web application that combines its own content with untrusted third-party advertising in the same browser instance (e.g., WebView). When the platform gives native-access rights to this WebView object, it effectively grants them not only to the application's own code, but also to unauthorized third-party code sharing the same WebView.

We demonstrate several security vulnerabilities caused by the unintentional delegation of native-access rights on the existing platforms. These vulnerabilities have not been reported in previous work [20]. For example, a malicious iframe included into an Ubuntu HTML5 app can use the privileges of the browser's main frame to steal the user's OAuth credentials for Gmail, Facebook, etc. Similarly, a malicious iframe in a Windows Runtime app can get unfettered access to geolocation. Google Chrome has a different but related problem, where messages sent by the app's local code to the app's Web code can be intercepted by malicious third-party content, thereby enabling content injection attacks.

We then design, implement, and evaluate POWERGATE, a new access-control mechanism for Web-based system applications. In contrast to the existing mechanisms, POWERGATE provides both

WWW 2015, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3469-3/15/05.
http://dx.doi.org/10.1145/2736277.2741663.

security and consistency by defining and correctly enforcing uniform access-control semantics for all native objects.

POWERGATE enables application developers to express access restrictions on native objects in terms of explicit principals such as "application's own local code," "application's own remote code," and "third-party remote code." Crucially, these policies do not rely on the risk-prone delegation of access capabilities to WebViews and other system components.

We implemented a prototype of POWERGATE based on Firefox OS. Our evaluation shows that the performance overhead of POWERGATE is very small vs. stock Firefox OS. We also demonstrate how POWERGATE can automatically emulate the access-control semantics of conventional Ubuntu, Google Chrome, Windows, and Firefox OS, thus ensuring 100% backward compatibility: any existing Web-based system application will run on a POWERGATE-enabled platform without any changes to the application's code.

2 Software stack

The software stack for Web-based system applications is shown schematically in Fig. 1. All platforms in our study implement some version of this stack; individual variations are explained in the corresponding sections. We divide the layers of this stack into two categories: those managed by the platform OS and those managed by individual applications.

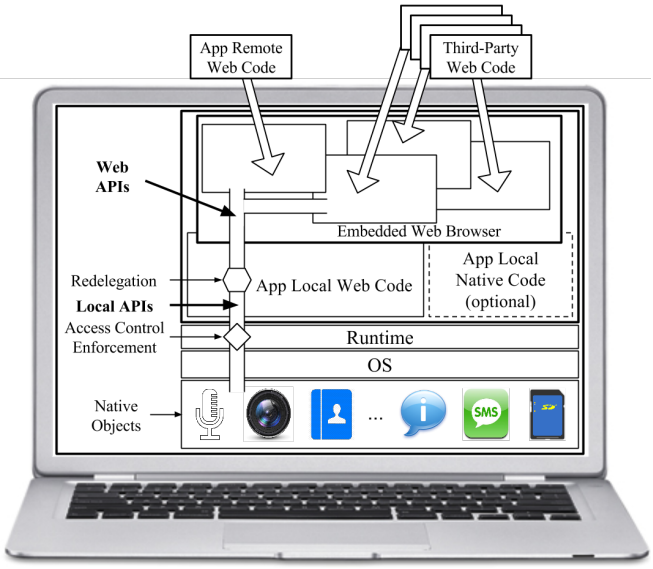

Figure 1: Software stack of Web-based system applications

2.1 Platform components

Native objects. These are the native resources and functionalities that the platform makes available to Web-based system applications. Depending on the platform, native objects may include hardware components (e.g., camera or GPS), special files (e.g., contacts list on a mobile device), and platform-level functionality (e.g., notifications or authentication credentials).

Local API to native objects. To enable applications to access native objects, each platform provides a runtime environment with special APIs. Different APIs may be available depending on the application's level of privilege. Even on platforms where local APIs are implemented in Web languages such as JavaScript, the platform may make them available only to the application's local code, i.e., the part already installed on the machine, but not to the remote Web code running inside a browser instance.

Web API to native objects. In contrast to the local API, Web API to native objects is available to the application's remote Web code. Web API is implemented via special browser objects or other mechanisms that enable Web content executing inside a browser to make calls outside the browser sandbox—either to the application's local code, or directly to the platform. Often, the Web API is a subset of the local API.

2.2 Application components

Manifest. The application's manifest lists what the application needs from the platform. The key part of the manifest is an explicit list of native objects that the application wants to access. In all platforms in our study, this list must be approved by the user when the application is installed. The platform's runtime environment is responsible for ensuring that an application cannot access a native object via the local or Web API unless this object has been approved by the user as part of the manifest.

Local code. Web-based system applications typically have a component which is installed on the host machine as a local program.

In the simplest case, the local component consists solely of the application's manifest and content caches, but in many applications it also includes executable code. This local code (1) creates embedded browser instances that execute the application's remote Web code (see below), (2) "links" the platform's local API and the browser's Web API so that the application's remote code, too, can access native objects, and/or (3) provides some offline functionality. Depending on the platform, the local code may be implemented in a Web language such as JavaScript, but it runs outside the browser, is not subject to the standard browser sandboxing policies, and has direct access to the platform's local API.

Own remote code. The bulk of the code of a Web-based system application is typically hosted at a remote, developer-controlled domain (we call it the app's *Web-home* domain). Hosting application code remotely allows developers to re-use the same code across multiple platforms, update it without involving the users, and dynamically generate new content.

This code runs on the user's machine inside an embedded browser instance (e.g., WebView) created by the application's local code. Consequently, it can access native objects via the Web API but not via the platform's local API.

Third-party remote code. Web-based system applications may also incorporate third-party Web code from origins other than the developer's own domain. For example, an advertising-supported application may include an iframe which dynamically fetches ads from different origins while the application is executing.

2.3 Threat model

For the purposes of this paper, we assume that the hardware, operating system, and the application (both the local and remote code written by the application developer, as well as the developer-specified privilege separation policies) are trusted. The main threat is third-party remote content included into the application. The origins of this content are not known statically, e.g., due to syndicated and auction-based advertising. It may be malicious and attempt to gain unauthorized access to camera, file system, etc.

The security of the software stack shown in Fig. 1 thus fundamentally depends on the platform's correct separation between the privileges granted to the application's local code, application's own remote code, and third-party remote code.

3 Existing platforms

Web-based system applications need to expose native objects to parts of the application's code (e.g., Web code hosted at the de-

veloper's own site) while hiding them from other parts (e.g., third-party ads). Standard OS access control works at the granularity of individual applications and cannot provide the required privilege separation. Platforms supporting Web-based system applications thus implement new access-control mechanisms for native objects.

In this section, we analyze the access-control models and enforcement mechanisms in Ubuntu, Google Chrome, Windows, and Firefox OS (summarized in Table 1). We show that native-access semantics are inconsistent across the platforms, explain the pros and cons of each approach, and demonstrate several security problems caused by the design flaws in access-control models.

3.1 Ubuntu

Overview. Ubuntu provides isolation between the app's local and remote Web code and also enforces the same origin policy on the remote Web code from different origins. The app's local code is treated as a distinct origin. Ubuntu uses a mix of install-time and runtime access control to protect access to native objects.

Figure 2: Ubuntu HTML5 apps: software stack

Privilege hierarchy. Ubuntu supports two classes of applications: HTML5 apps and regular Web apps. The latter execute inside a Web browser and have access to a very limited set of native objects. By contrast, HTML5 apps execute inside a Web "container" without the browser UI elements. The overall architecture of Ubuntu HTML5 apps is shown in Fig. 2.

Ubuntu supports three different API classes for HTML5 apps: Ubuntu platform APIs (Unity), W3C APIs, and Cordova APIs. Ubuntu platform APIs provide access to OS services such as notification, messaging, media player, launcher, etc. The W3C APIs follow W3C specifications and provide access to a very limited set of device resources such as geolocation. The Cordova APIs [10] provide access to a wide range of OS resources such as accelerometer, camera, capture, contacts, device, file, geolocation, and storage. Cordova uses a plugin framework to expose device-level functionality via a JavaScript interface. Ubuntu requires app developers to either package their own copies of the Cordova runtime with the app, or use the one offered by the platform.

Principals and access-control implementation. Ubuntu uses the open-source AppArmor utility to sandbox each app. An app developer can either use pre-existing templates to configure the sandbox for the resources her app needs, or customize the templates by adding her own set of "policy groups," roughly analogous to Android permissions. Sample policy groups include accounts, audio, calendar, camera, connectivity, contacts, and location, etc. [3]

The Cordova API is accessible to Web code from any origin [20]. Even though Cordova supports origin whitelisting, Ubuntu does not expose it to app developers, thus there is no way to restrict native-object accesses to a particular origin(s). On the other hand, Ubuntu ensures that most of the Unity API is only accessible to the app's local code and not to any Web code. Unfortunately, this policy has exceptions which lead to serious security vulnerabilities.

Pros. Ubuntu relies on AppArmor and hides most of the fine-grained, origin-based access control from the user.

Cons. As mentioned above, Ubuntu exposes the Cordova API to all Web origins, allowing untrusted Web code to access camera, microphone, etc. [20] Furthermore, Ubuntu lets users link their online accounts such as Google and Facebook to their OS account. Once linked, the login credentials are accessible via the OnlineAccounts API [36]. Even though untrusted Web code inside iframes cannot query this API directly, it can navigate the top frame to its own origin, steal the user's OAuth credentials via the API (since access is granted to a particular *frame*, not to a particular Web origin), and redirect the main frame back to the original page to keep the attack stealthy. This attack enables malicious Web content to hijack users' Google and Facebook accounts.

3.2 Chrome

Overview. The software stack of Google Chrome apps is shown in Fig. 3. Chrome provides isolation between the app's local code, which it treats as a distinct origin different from all Web origins, and remote content, to which it applies the same origin policy. Chrome uses install-time access control for the native objects declared in the app's manifest, but developers can add stricter runtime checks by coding them in the app's JavaScript shim.

Chrome also supports optional permissions,[1] which are declared in the app's manifest but must be approved by the user at runtime when the object is accessed. Optional permissions are not the focus of this paper because they apply only to a handful of native objects, not including the more sensitive ones such as camera.

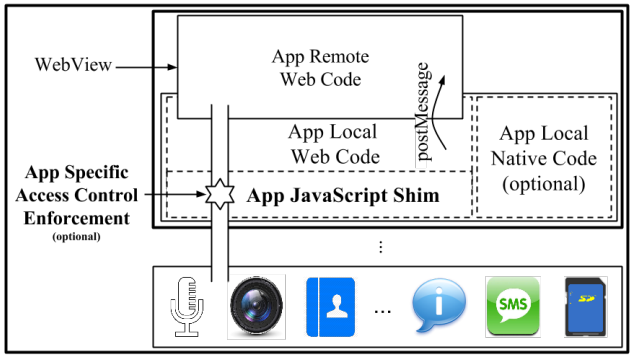

Figure 3: Chrome apps: software stack

Privilege hierarchy. Google Chrome supports three different types of applications. *Hosted apps* reside on a remote server, with only a manifest file and an icon installed on the user's machine. Hosted apps have access only to "background", "clipboardRead", "clipboardWrite", "geolocation", "notifications", and "unlimitedStorage" objects. *Chrome apps* run inside their own browser instances but without browser UI. *Extensions* run inside regular browser instances but can integrate more closely with the browser and bypass the same origin policy. Chrome apps and extensions have access to most of the native objects.

Principals. The local Web code of Chrome apps cannot include third-party resources such as scripts and embedded objects. Unlike Firefox OS (described below), Chrome does not allow iframes pointing to third-party content. Instead, such content must be put into WebView tags. Unlike iframes, these WebViews are separate browser instances, thus their rendering engines run in separate processes. Even if malicious content compromises the rendering pro-

[1] `https://developer.chrome.com/apps/permissions`

Platform	Privilege hierarchy	Install-time access control	Runtime access control	Each tuple defines a principal
Ubuntu	Yes (2 levels)	Yes	Yes	(app, local/remote code)
Chrome	Yes (3 levels)	Yes	(optional) in JavaScript shim	(app, local/remote code, Web origin*)
Windows	No	Yes	Yes	(app, local/remote code)
Firefox OS	Yes (4 levels)	Yes	Yes	(app, browser instance, Web origin)

Table 1: Access control enforcement on different platforms.
*Code for origin-based access control enforcement is the responsibility of the app developer.

cess of its WebView, the other browser instances are still protected by the OS-level process isolation.

By default, native objects in Chrome are exposed only to local app code, but developers can implement JavaScript shims that expose certain objects via Web API to the remote code executing inside WebView instances. The developer is responsible for ensuring that this access is not abused by untrusted third-party code executing in the same WebViews. This is a problem because, when enabled, a WebView API to a native object can be accessed by any content inside this WebView, regardless of its origin.

Access-control implementation. Each Chrome app must request via its manifest the permissions for all native objects it accesses [9]. Sensitive permissions (videoCapture, audioCapture, etc.) are shown to the user for approval when the app is installed. Once approval is granted, the app can access all requested native objects without any further user interaction.

Pros. Chrome hides its fine-grained, origin-based isolation from the user. Individual Web APIs are not activated by default, but must be explicitly requested by the app.

Cons. Conceptually, Chrome relies on delegating native-access rights not to named principals (explicitly specified Web origins) but to system components (WebView instances). This is risky because a given component may include code from untrusted principals. Security thus completely depends on the developers correctly implementing origin checks for each resource request coming from a WebView instance. These checks can rely on *WebRequestEventInteface*,[2] or use various match patterns.[3] In either case, the access-control code is subtle and difficult to write correctly [43]. The sample WebView app distributed by Google Chrome via GitHub [8], which is supposed to serve as a coding guidance for other apps, is missing any such checks and consequenly exposes the native objects to all Web origins.

Furthermore, even correctly implemented checks can be bypassed. For example, the current implementation of the "pointer lock" object incorrectly reports the source of the request to be the origin of the main frame even when the request comes from inside an iframe, rendering all origin checks ineffectual.

Another problem inherent in Chrome apps is their use of HTML5 postMessage for communication between the app's local Web code and its remote Web code. By design, the remote Web code running inside WebView does not have a reference to the parent app's window. It must first receive a postMessage from the local side of the application before it can send a message to it. Unfortunately, the local code of some existing Chrome apps (e.g., Lucidchart Diagrams) sends postMessage with "*" as the destination, i.e., without restricting the receiver's origin. Malicious third-party Web code running inside an iframe can "frame-bust" (i.e., navigate the main frame of its WebView to the attacker's URL), capture the message and its contents, and change the main frame to its old URL. Once it obtains a reference to the parent app's window in this way, malicious code can also send messages to the app's local code. This makes content injection attacks possible.

[2] https://developer.chrome.com/apps/tags/webview.html#type-WebRequestEventInteface

[3] https://developer.chrome.com/apps/match_patterns.html

3.3 Windows

Overview. Windows 8.1 supports Windows Runtime apps written in HTML5 and JavaScript. These apps have access to native objects such as camera, removable storage, and geolocation via the Windows.* API. Remote Web content included in an app cannot access the namespace of these native objects directly. The overall architecture of Windows Runtime apps is shown in Fig. 4

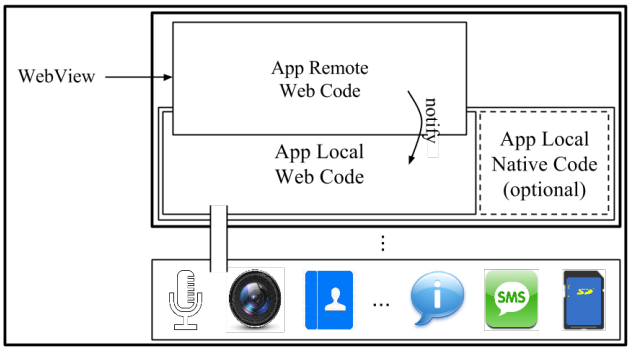

Figure 4: Windows Runtime apps: software stack

Principals and access-control implementation. A Windows Runtime HTML5 application can include remote Web content in two ways: in an iframe embedded into an internal application page or in a WebView object. If remote content is loaded in an iframe, the developer must whitelist its domain in order for the content to be fetched and loaded successfully. Windows allows only HTTPS domains to be whitelisted. No whitelisting is needed if the content is to be displayed inside WebView. Windows also requires that if the main WebView frame is fetched over HTTPS, all inner iframes must be fetched over HTTPS as well.

Windows uses 20 capability groups [51] to protect native objects. Developers must declare all capabilities needed by their apps in the manifests. The user is prompted when an app wants to use a given permission for the first time; all subsequent requests are granted or denied based on this decision.

The permission prompt shows the name of the app but not the origin of the request (e.g., "Can TestApp use your location?"). This does not affect native objects managed by Windows Runtime because they are not visible to remote Web code, but it affects the W3C geolocation API which is exposed to Web code by Windows. If an internal application page requests access to the geolocation object and the user approves, all third-party Web content included in this app will implicitly get access to geolocation.

Pros. Windows exposes access to the Windows Runtime library only to local application code, thus reducing the security risks of running untrusted remote Web code in embedded browser instances.

Cons. Windows does not provide origin-based access control for native device resources. This access-control model is inadequate

for objects such as geolocation that are available through both Windows Runtime and W3C API. As a result, malicious third-party content can surreptitiously obtain the user's location.

Due to lack of built-in mechanisms for intra- and inter-principal communications, developers of Windows Runtime apps must rely on custom schemes—for example, the app's Web code can use the *window.external.notify()* API to talk to the app's local code outside the browser. The domain of this Web code must be whitelisted; if there are multiple domains in the whitelist, then each of them can send a *window.external.notify()* message. In this case, the developer must write (error-prone) code that checks *sender.callingUri* and disambiguates calls made by different senders.

3.4 Firefox OS

Overview. Firefox OS supports fine-grained access control for native objects. Principals are defined by the application instance, embedded browser instance, and Web origin [16]: each combination of these has its own distinct access rights. Firefox OS uses a mix of install-time and runtime, prompt-based access control.

The software stack of Firefox OS applications is shown in Fig. 5. The OS part of the stack (not shown in detail in the picture) consists of three major components: Gaia front-end responsible for maintaining the UI, the underlying operating system layer called Gonk that provides abstractions for accessing the device hardware, and an intermediate layer called Gecko which includes the runtime engines for HTML, CSS, and JavaScript and is responsible for enforcing all security policies regarding access to native objects.

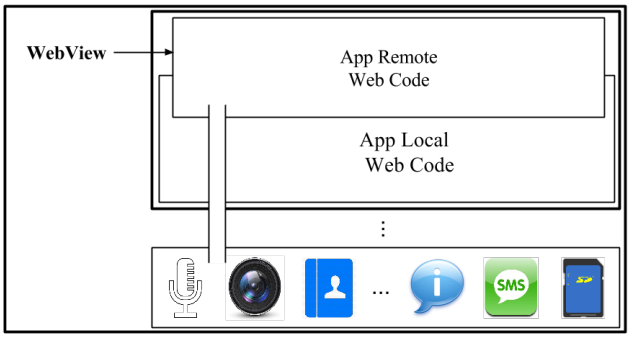

Figure 5: Firefox OS apps: software stack

Privilege hierarchy. There are two types of applications (apps) in Firefox OS: hosted and packaged. *Hosted apps* have no local content and their manifest file is hosted at a remote server. This file may also contain instructions for caching or offline usage of the app. Isolation of hosted apps is based on the Web origins [4] of their manifest files, defined by the protocol, domain, and port number. Firefox OS allows only one hosted app per origin since it cannot isolate distinct hosted apps if they have the same origin. To prevent arbitrary files from being interpreted as manifests, Firefox OS requires manifest files to be served with a specific MIME type ("application/x-web-app-manifest+json").

Unlike hosted apps, packaged apps store local content on the user's machine during installation. Packaged apps can be further categorized into three categories, in increasing order of privilege: regular, privileged, and certified apps.

Regular packaged apps have the same privileges as hosted apps. The only difference is that they can work offline because their contents are stored locally on the user's machine.

Privileged packaged apps contain an explicit list of assets in a ZIP archive, reviewed and signed by the app store. All assets are verified during installation and stored locally.

Certified packaged apps are special apps with elevated privileges, similar to setuid binaries in UNIX. In contrast to privileged apps, almost all permissions for certified apps are implicit and not controllable by the user. Certified apps are signed with the same key as the OS and distributed solely by the device manufacturer.

Principals. As mentioned above, each tuple of (application instance, browser instance, Web origin) defines a unique access-control principal. Firefox OS allows different browser instances to coexist inside a single application. The application can use them to render Web content from the same origin, yet prevent the sharing of cookies and other browser state.

The local content of an application is treated by Firefox OS as a special principal whose origin is different from all Web origins. All application code is written in HTML5 and JavaScript; Firefox OS applications do not have local native code. The application's local code cannot include content from other origins via tags like $embed$ or $script$. It can include third-party content using $iframe$, but in this case the included content is isolated from the local content by the same origin policy [4] since its origin is different from that of local content. Firefox OS uses content security policies (CSP) to enforce these restrictions.

Access-control implementation. Depending on the application's privilege level (i.e., hosted, regular, privileged, or certified), certain permissions are implicit and the rest are explicit; also, certain objects are not available to certain classes of applications. For example, contacts and device storage API are not available to regular applications. The application developer must list all permissions the application needs, whether explicit or implicit, in its manifest. When enumerating individual permissions in the manifest, the developer must supply a short description of why the application needs this permission and how it will be used. This description is displayed to the user when the user is asked to authorize access to the underlying object and on other UI screens concerning the application's management and integration with the OS.

Once the application is installed, the implicit permissions are granted without any user interaction. The explicit permissions generate user prompts when the corresponding objects are requested. The user grants or denies access and can optionally instruct the system to remember her decision for subsequent requests of the same type coming from the same principal.

Local APIs to some native objects have an additional "access" property, which lets the developer specify the access rights he wishes the application to have with respect to the corresponding object, such as "readonly", "createonly", "readcreate", and "readwrite". These properties are specified in the manifest but not shown to the user. Firefox OS lets users monitor permission state and change applications' permissions via special UI. More details about the Firefox OS permission system can be found in [15].

Pros. The access control mechanism in Firefox OS helps developers implement flexible, fine-grained isolation between application components without writing any extra code.

Cons. Firefox OS bombards the user with permission requests for each (application, browser, origin) tuple. For example, suppose an app requests access to geolocation from both its local HTML5 code and remote HTML5 code hosted at the developer's domain. The user will be prompted twice: once with the name of the app and second time with the domain name. To make matters more confusing, suppose that this app uses two different browser instances, both of which render content from the same domain that needs access to geolocation (e.g., both show the same geotargeted ad). In this case, the user will be prompted twice with the same domain name.

4 Design of POWERGATE

The purpose of POWERGATE is to help developers of Web-based system applications write access-control and privilege separation policies that (1) support inclusion of untrusted third-party content such as Web advertising, and (2) restrict this third-party content to an explicitly enumerated subset of native objects.

The common case in all existing Web-based system applications is that the application requests native access only for its own code, which may be hosted on the device itself and/or provided from the application's remote Web server. All third-party code included into these applications is conventional Web content such as ads and "Like" buttons, intended for conventional browsers without native-access APIs. Therefore, no third-party code—even if included into the application's own code (for example, as a syndicated ad in an iframe)—should be able to access native objects, except possibly geolocation which may be needed for geotargeted advertising.

To support this common case, POWERGATE gives developers an easy way to "name" all external domains so that they can be blocked en masse from native access. For applications with more fine-grained native-access policies, POWERGATE can also support different rights for different third-party domains. This takes a single line per domain in the application's policy specification.

Access-control principals. POWERGATE policies are expressed in terms of native objects and access-control principals, as opposed to the platform components (shims, browser instances, etc.). We chose this approach because the distinction between the application's own code and third-party code does not map neatly onto the platform components. For example, on some platforms the application's own code is split into a local part hosted on the user's machine and a remote part hosted on the developer's server. At the same time, certain components such as embedded Web browsers combine the application's own code with third-party code.

Web origins are natural access-control principals in many scenarios—for example, when implementing privilege separation inside pure Web applications [2, 29]—but they are often too fine-grained for developers' purposes. Consider the developer of an ad-supported application who wants to allow all ads in her application to access the user's location for better geotargeting. In the case of syndicated or auction-based advertising, the origins of these ads are determined dynamically and enumerating them in advance as part of the access-control policy is not feasible.

There are four main principals in POWERGATE: *application's local native code*, *application's local Web code*, *application's remote Web code*, and *third-party Web code*. In Section 6.1 we show that these principals are sufficient to emulate the access-control models of all existing platforms. Optionally, individual domains can be specified as sub-principals of "third-party Web code."

The developer specifies a static "Web-home" domain hosting the application's remote Web code. All other Web origins are treated as sources of untrusted code and thus—for the purposes of native access only—are clustered into the "third-party Web code" principal. This lumping does not introduce any security vulnerabilities because the same origin policy is still enforced by the browser, thus individual Web origins within the "third-party Web code" principal still cannot access each other's Web resources. Furthermore, by default, this joint principal is blocked from accessing any native objects. Therefore, third-party Web origins have strictly fewer rights when they are combined into this joint principal than they have individually on any existing platform.

Access-control policies. In a POWERGATE policy, every native object is associated with an ACL-style list of privileges for each principal. POWERGATE has a very simple hierarchy of privileges. The default setting is *deny*: the principal has no access to the object. A principal may be *allowed* to access the object unconditionally, or the system may *prompt* the user for permission. For complex native objects (e.g., contacts, external storage, etc.), POWERGATE supports the access qualifiers of *readonly*, *readwrite*, *readcreate*, and *createonly*, same as stock Firefox OS.

The app's remote Web code is associated with a specific Web origin which, by definition, includes the protocol (HTTP or HTTPS). If the protocol is not specified in the manifest, POWERGATE assumes HTTPS. If access to a particular native object is allowed unconditionally for third-party Web code, then POWERGATE restricts it to HTTPS origins to prevent network attacks.

This simple system of principals and privileges can express rich, yet conceptually straightforward policies. For example, the application's local native and Web code may access the contacts, videos, and pictures, the application's remote Web code may access only videos and pictures, and third-party code may access geolocation but only if the user explicitly approves.

To enable secure communication between the app's local Web code and its remote Web code, POWERGATE introduces new API functions *sendToAppLocalWebPrincipal(String msg)*, *receiveFromAppLocalWebPrincipal()*, *sendToAppRemoteWebPrincipal(String msg)*, and *receiveFromAppRemoteWebPrincipal()*. Messaging between other pairs of principals can be supported in a similar fashion, if needed. These new messaging APIs deliver messages to explicitly named principals rather than individual system components. Therefore, applications that use them do not suffer from the vulnerabilities affecting current Web-based system applications whose intra-application communication is based on HTML5 postMessage. In many such applications, the recipient's origin is specified as *"*"* and messages can thus be intercepted by any third-party code running in the same browser.

Centralized enforcement and backward compatibility. All access-control decisions in POWERGATE are made in a single place in the code. Access-control logic is independent of the actual native object being accessed and does not depend on the platform-specific implementation details. POWERGATE is thus "future-proof": it protects access not only to all native objects in today's platforms, but also to any objects added in future releases.

POWERGATE ensures that existing applications behave as before when ported to POWERGATE. In Section 6.1, we show how the manifest of any existing application can be rewritten to emulate the access-control semantics of any of the platforms described in Section 3. In Section 6.2, we show that all existing Web-based system applications can run without any code modifications on a POWERGATE-capable platform and, if needed, preserve the same access-control semantics they currently have.

Limitations. By design, POWERGATE only protects access to native objects. To protect and isolate Web resources such as DOM, POWERGATE relies on the Web browser's implementation of the same origin policy. Like all other OS and browser access-control policies, POWERGATE's policies are binary: access is denied or allowed, but there is no notion of "rate-limiting" (e.g., allowing an app to access a certain object only 5 times). We did not observe the need for such richer policies in the existing apps.

Remote principals are defined by their Web origin or a set of origins, since this is the only information about them that is readily available to the host platform. POWERGATE thus cannot express policies such as "only domains belonging to local businesses can access geolocation" because there is no feasible way for the platform to determine whether a particular domain truly belongs to a

local business. This is a generic limitation of the entire Web model and not specific to POWERGATE.

POWERGATE does not control what happens to information after it has been accessed by an authorized principal. For example, the user may grant access to the contacts list to the application's own code, which then sells this data to an advertiser. This is a generic limitation of host-based access control and cannot be solved without radically restructuring the existing host and network infrastructure to enforce Internet-wide information flow control.

5 Implementation of POWERGATE

The design of POWERGATE is platform-independent, but our proof-of-concept prototype implementation is based on Firefox OS. Implementations on other platforms would be essentially identical since all of them, except Chrome, have permission managers that are similar to Firefox OS and can be modified identically. In Chrome, native access is controlled inside the app, not the OS, thus POWERGATE would be provided as a library. The alternative is to add a Firefox-OS-style permission manager to Chrome.

The key data structure in POWERGATE is *PowerGateStore*, an in-memory hash map indexed by app id which stores the access privileges of currently running apps. We also use the same term for the file from which this hash map is populated when a new app is launched. The in-memory *PowerGateStore* is used to make access-control decisions at runtime, whereas the *PowerGateStore* file is used to persist apps' access-control settings between host restarts. *PowerGateStore* is managed internally by the OS and cannot be accessed by applications unless the OS is compromised, which is beyond our threat model.

The implementation of POWERGATE consists of two parts. The first part parses the app's manifest when the app is installed and updates the *PowerGateStore* file accordingly. The second part decides at runtime whether to unconditionally allow, prompt the user for permission, or deny a given access request to a native object.

At app installation time. POWERGATE parses the app's manifest file and creates a *Preference* object for each (native object, principal) pair declared in the manifest. This object's string representation is prefixed with the app's id, which is assigned by the underlying system at install time. A *Preference* object is also created for the app's own Web-home domain.

The set of all *Preference* objects for the app being installed is then stored in the *PowerGateStore* file. For example, if the app's system id is 32, its Web-home domain is `www.example.com`, and the app allows only Web code from `adserver.com` to access the *geolocation* object (with explicit user approval), then POWERGATE will create the following entries in *PowerGateStore*:

```
32.webhome: https://www.example.com
32.geolocation.third-party-web-code.prompt.true.domain:
     https://adserver.com
```

The user can manually overwrite these policies, if needed.

At app runtime. The in-memory representation of the access-control policies of all apps currently running on the host is kept in the *PowerGateStore* hash map, keyed by app id. Each entry in this map includes the app's Web-home domain and a *Policy* object defining the policies for all native objects to which the app has requested access via its manifest file. These *Policy* objects are hash maps keyed by the native object's name and enumerating the access rights of each principal.

Access-control policies. The possible values for the *principals* tag in a policy are *local-native-code*, *local-web-code*, *app-web-code*, or *third-party-web-code*. The current version of Firefox OS does not allow applications to include native code, thus all existing applications are implemented exclusively in Web languages such as HTML5, JavaScript, and CSS. On other platforms, however, applications can include local native code, necessitating support for the *local-native-code* principal. Optionally, individual domains can be specified as sub-principals of the *third-party-web-code* principal.

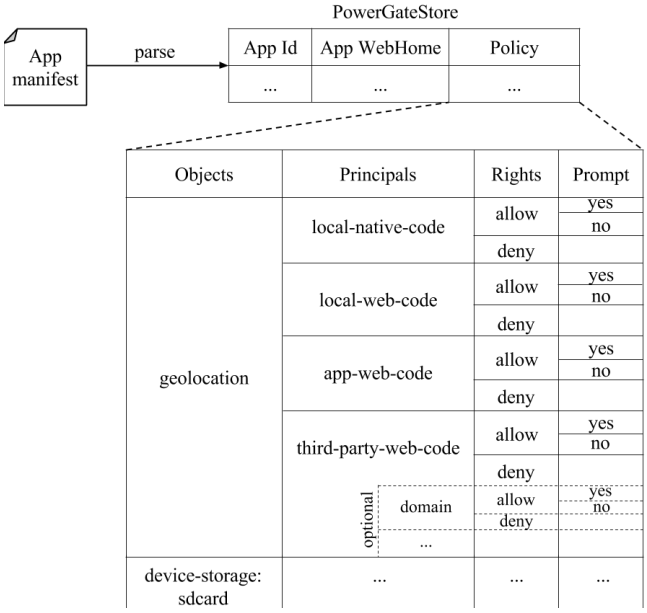

Figure 6: PowerGateStore

The *prompt* tag allows developers to specify whether the user should be prompted for permission when access to a given native object is attempted. The default value of this tag is *"yes"*. In POWERGATE, the user can instruct the GUI to remember her choice. By explicitly incorporating user prompts into access-control policies, POWERGATE can emulate the following access-control semantics: *prompt: no*, *prompt: always* (the user instructs the system to *not* remember her choice), *prompt: first-time-use* (the user instructs the system to remember her choice).

Access-control enforcement. POWERGATE's enforcement mechanism is shown schematically in Fig. 7. It is implemented in 366 lines of C++ code. Specifically, in *nsPermissionManager.cpp*, POWERGATE modifies the internal implementation of

```
CommonTestPermission(nsIPrincipal * aPrincipal,
                     const char * aType,
                     uint32_t   * aPermission,
                     bool  aExactHostMatch,
                     bool  aIncludingSession)
```

with a function call to:

```
uint32_t * aPermission
PowerGate::GetAction(uint32_t appId,
                     const char * aType,
                     string prePath)
```

defined in *PowerGate.cpp*. Since the *CommonTestPermission()* function is the central location used by Firefox OS to make access-control decisions about any native object, *PowerGate::GetAction()* effectively assumes this responsibility.

In POWERGATE, the decision to allow or deny access is based on the app id, the name of the object being requested, and the requesting principal. The app id and the principal are determined from the information provided by the platform's runtime environment for every request. The name of the object is conveyed by the platform's local API. Specifically, POWERGATE derives the values of *appId* and *prePath* from the *aPrincipal* object and uses *aType* as the native object name. POWERGATE does not use *ExactHostMatch* be-

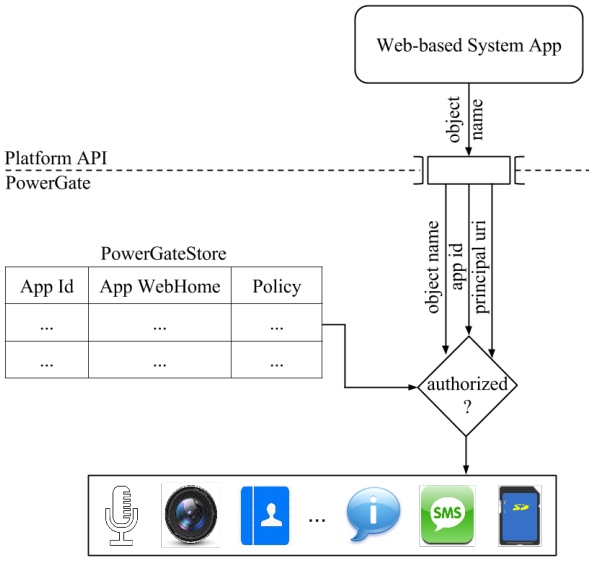

Figure 7: Access-control enforcement at runtime

cause it always assumes this flag to be true when the request comes from the app's Web code or a third-party origin explicitly defined in the app's manifest, and false in all other cases. POWERGATE then looks up the app id in *PowerGateStore*. If a match is found, POWERGATE retrieves the app's policy object. POWERGATE internally computes the value of *aIncludingSession* based on the prompt part of the policy and/or user's choice (if the user was previously prompted). Finally, POWERGATE uses the policy to decide whether to allow the request, deny it, or prompt the user.

Inter-principal communication. To enable secure communication between the app's local Web code and its remote Web code, POWERGATE provides a new messaging API. This API can be implemented via a new, distinct native object, but our current proof-of-concept prototype implementation is based on a sandboxed variant of the existing *Alarms* object, which we called *Communicator*. When the local Web code wants to talk to its remote counterpart, it invokes *sendToAppRemoteWebPrincipal(String msg)*. The local Web code then sends a *postMessage("pending message", "*")* to the browser instance executing remote code to notify the latter that a message is waiting for it. Upon receiving this notification, the remote Web code retrieves the pending message by invoking *receiveFromAppLocalWebPrincipal()*. Note that there is no need to protect the content of the postMessage since it does not contain any sensitive information. Even if malicious third-party code intercepts this postMessage, the POWERGATE access-control mechanism will prevent it from retrieving the actual message since it is not the *app-web-code* principal. Messaging from the app's remote Web code to the app's local Web code works similarly by using the *sendToAppLocalWebPrincipal(String msg)* and *receiveFromAppRemoteWebPrincipal()* API.

6 Evaluation

For the evaluation of the existing platforms described in Section 3, we used a Dell Latitude E6500 laptop executing Windows 8.1 with Windows Phone Simulator; a MacBook Pro laptop executing OS X 10.9.4 with Google Chrome 34.0.1847 and two virtual machines installed inside VMware Fusion 5.0.4, running Ubuntu 13.10 and Ubuntu 14.04, respectively; and a Dell PowerEdge 2950 running Ubuntu 12.10 LTS and the Firefox OS version 1.3 simulator. To evaluate our proof-of-concept implementation of POWERGATE, we modified the Firefox OS simulator with scripts that automate user interaction and thus generate deterministic sequences of native-access requests. All performance evaluation of POWERGATE was done on the Dell PowerEdge 2950 running Ubuntu 12.10 LTS.

6.1 Backward compatibility

POWERGATE can emulate the access-control policies of all four platforms analyzed in our study. This is done by automatically rewriting the application's manifest, without any changes to the application's code.

Ubuntu. In Ubuntu HTML5 apps (see Section 3.1), local code has access to the Unity API [49], allowing apps to reserve a spot on the Launcher or integrate with Messaging and Music menus. This API is not available to either the app's own, or third-party Web code. The corresponding POWERGATE policy is as follows:

```
In Ubuntu:
  Allowed

In PowerGate:
  "policy_groups": [
    "unity": {
      "principals": {
        "local-web-code": {"prompt": "no"}}}]
```

For objects that are likely to be useful to HTML5 apps, such as the *OnlineAccounts* object, Ubuntu grants access (without user interaction) to all Web content loaded in the main frame of the app's WebView, but not to third-party content loaded in iframes. This corresponds to the following POWERGATE policy:

```
In Ubuntu:
  "policy_groups": ["accounts"]

In PowerGate:
  "policy_groups": [
    "accounts": {
      "principals": {
        "local-web-code": {"prompt": "no"},
        "app-web-code": {"prompt": "no"}}}]
```

Cordova on Ubuntu (see Section 3.1) comes with a set of 16 plugins [10], enabled by default. Access is implicitly granted to all Web content regardless of its origin [20]. In POWERGATE-enabled Ubuntu, this policy is implemented by instrumenting each plugin's native object with the appropriate access attributes. For example, the policy for the camera object is expressed as:

```
In Ubuntu:
  "policy_groups": ["camera"]

In PowerGate:
  "policy_groups": [
    "camera": {
      "principals": {
        "local-web-code": {"prompt": "no"},
        "app-web-code": {"prompt": "no"},
        "third-party-web-code": {"all": {"prompt": "no"
            }}}}]
```

Chrome. In Chrome (see Section 3.2), the application's local Web code—but not its remote code—has access to the local API. Chrome does not require that application developers declare their intention to use the local API in the manifest. In POWERGATE, a new object called "locals" represents the local API in Chrome. The declaration in the manifest is as follows:

```
In Chrome:
  Allowed

In PowerGate:
  "locals": {
    "principals": {
        "local-web-code": {"prompt": "no"}}}
```

Using its access to the local API, the application's local Web code can enable Web API for individual native objects. If this API is enabled, the application's remote Web code—as well as any third-party Web code in the same browser—can use it to access the corresponding object. For instance, the access policy for the video capture object can be expressed as:

```
In Chrome:
  "permissions": {"videoCapture"}

In PowerGate:
  "permissions": {
    "videoCapture": {
      "principals": {
        "local-web-code": {"prompt": "no"},
        "app-web-code": {"prompt": "no"},
        "third-party-web-code": {"all": {"prompt": "no"
             }}}}}
```

Windows. In Windows Runtime apps (see Section 3.3), only local application files have access to the Windows.* native objects and the user is prompted on the first use of each native object [51]. Therefore, in POWERGATE-enabled Windows, the access policy for each object is listed individually. For instance, the manifest entry for the camera object is expressed as:

```
In Windows:
  <DeviceCapability Name="webcam" />

In PowerGate:
  <DeviceCapability Name="webcam"
    Principal="local-native-code" Prompt="Yes"/>
  <DeviceCapability Name="webcam"
    Principal="local-web-code" Prompt="Yes"/>
```

Firefox OS. In Firefox OS (see Section 3.4), all requests to native objects can be categorized into four categories: *denied*, *granted*, *prompt: first-time-use*, and *prompt: always*. While different native objects have different access policies [15], they can all be expressed in POWERGATE's terms using one of the *local-web-code*, *app-web-code*, or *third-party-web-code* principals. For example, here is the policy for the desktop-notification object:

```
In Firefox OS:
"permissions": {
  "desktop-notification": {
    "description": "Notify the user.", }}

In PowerGate:
"permissions": {
  "desktop-notification": {
    "description'': "Notify the user.",
      "principals": {
        "local-web-code": {"prompt": "no"},
        "app-web-code": {"prompt": "yes"},
        "third-party-web-code": {"all": {"prompt" : "yes
             "}}}}}
```

6.2 Deployment

By design, POWERGATE augments the application's manifest on all platforms with information about the application's Web home and the access rights it grants to various principals with respect to the platform's native objects. To disambiguate between manifest formats, POWERGATE increments the version number of the manifest file by 1. For instance, the current version number of the manifest in Chrome apps is 2, whereas POWERGATE-compatible Chrome apps have manifest version 3. Similarly, current Ubuntu HTML5 apps have policy group version 1, whereas POWERGATE-compatible Ubuntu HTML5 apps have policy group version 2.

At application installation time, POWERGATE checks the version number of the application's manifest. If it matches POWERGATE's manifest version number for the given platform, the manifest is parsed as described in Section 5. Otherwise, POWERGATE assumes that the application is not aware of POWERGATE and cannot take advantage of its facilities. In this case, POWERGATE rewrites the manifest with appropriate settings to ensure backward compatibility with the original platform, as described in Section 6.1.

POWERGATE does not change the API used to access native objects and is thus fully transparent to all existing applications. Any application that currently runs on stock Firefox OS will run on POWERGATE, too. We developed a test suite and verified that in 100% of the tests, API access to native objects from legacy code returns exactly the same results with and without POWERGATE.

6.3 Performance

Micro benchmarks. Table 2 compares how long it takes to make access-control decisions in POWERGATE vs. stock Firefox OS on individual accesses to native objects. All numbers are in microseconds and averaged over 3000 runs.

	Firefox OS	PowerGate	Overhead
ALLOW_ACTION	39.76 μs	42.34 μs	1.0649X
DENY_ACTION	18.19 μs	18.65 μs	1.0253X

Table 2: Micro benchmark: Performance overhead per native-object access as visible to the Permissions Manager

Macro benchmarks. Table 3 compares the overall performance of POWERGATE with stock Firefox OS on individual accesses to native objects from JavaScript. All numbers are in milliseconds and averaged over 3000 runs.

	Firefox OS	PowerGate	Overhead
ALLOW_ACTION	66.5733 ms	67.3916 ms	1.0123X
DENY_ACTION	1.0527 ms	1.0533 ms	1.0006X

Table 3: Macro benchmark: Performance overhead per native-object access as visible to JavaScript

7 Related work

Origin-based access control for Web content. Origin-based access control [4] in Web browsers isolates content from different origins, defined by domain, protocol, and port. Zalewski [52] and Singh et al. [42] describe several flaws in browsers' implementations of origin-based access control.

Browsers conforming to the HTML5 specifications provide the postMessage API for communication between frames from different origins. Prior work [5, 43] found security flaws in cross-origin communication that can be used to bypass origin-based access control by exploiting incorrect authentication of the sender, certain frame navigational policies, etc..

Wang et al. [50] investigated cross-origin communication channels, such as intent, scheme, and Web-accessing utility classes, on Android and iOS, and found that they are vulnerable to cross-origin attacks from malicious apps and malicious links clicked by users. Their proposed defense, Morbs, labels every message with its origin and uses this information to enforce developer-specified security policies. In contrast to Morbs, which protects Web resources from malicious apps, POWERGATE protects native resources from malicious third-party content inside benign apps.

Origin-based access control for native-access APIs. Modern mobile platforms, including Android and iOS, allow applications to embed instances of Web browsers and equip them with interfaces or "bridges" that let Web code access resources outside the browser. For example, 'addJavascriptInterface' is used to add such interfaces to Android's WebView. In older versions of Android, these interfaces can be exploited by malicious Web code to compromise the host via reflection attacks [1, 34, 39]. Luo et al. [31] observed that

these WebView interfaces can be accessed by any script regardless of its origin. Chin et al. [7] found instances of this problem in several Android applications. Other papers [22, 44] investigated how Android advertising libraries expose device resources via 'addJavascriptInterface' to Web advertisements.

Hybrid application development frameworks such as PhoneGap and Web Marmalade use these bridges to equip apps written mostly in JavaScript and HTML5 with native access to the host platform. Hybrid apps are somewhat similar to Web-based system applications, but the native-access capabilities and the corresponding access control are provided by an application-level framework, not by the platform itself. Georgiev et al. [20] showed how untrusted Web content can bypass this access control in popular hybrid frameworks. Jin et al. [26, 27] demonstrated how naive uses of the framework-supplied functionality, such as reading from 2D barcodes and scanning Wi-Fi access points, can expose hybrid apps to code injection attacks. MobileIFC [41] leverages hybrid-framework APIs to enforce a fine-grained, information flow-based permission model that controls what an untrusted application can do with the data received through the framework API.

In this paper, we consider a different layer of the software stack and investigate the native-access APIs provided by the platform to the applications, not compiled into the application via a third-party library. The APIs we study (1) are not based on 'addJavaScriptInterace', (2) are exposed to code *outside* the Web browser, and (3) access control is performed by the platform, not by the application. Since the platform is responsible for access-control enforcement, defenses that are suitable for hybrid frameworks (such as NoFrak [20]) do not apply, necessitating modifications to the platform. In general, platform-level native access is more attractive for app developers than relying on hybrid frameworks because the developers do not need to update, repackage, and redistribute their applications every time the framework is updated. Furthermore, platform support obviates the need to include a copy of the framework in every application, thus reducing code bloat.

As an alternative to browser interfaces for native access, Gibraltar [28] exposes hardware resource via a Web server hosted on localhost and accessible via AJAX.

Isolating untrusted advertisements. Several privilege separation techniques aim to minimize the risks of untrusted advertisements included into Web applications [2, 29, 40, 44]. These approaches either isolate advertisements inside iframes, or else run them in separate processes and control communication between the application and the confined content using developer-specified policies. For example, AdSplit [40] puts Web-based advertisements into separate WebView instances and relies on the application to restrict invocations of privileged APIs from these WebViews.

A key purpose of Web-based system applications studied in this paper is to enable the developers to integrate already deployed Web applications with new Web code that accesses system resources. Web-based system applications are thus effectively mashups, with multiple principals executing together in the same browser instance. Applying solutions like [2, 29, 40, 44] requires re-engineering of the existing code. By contrast, POWERGATE does not require modifications to advertising-supported legacy Web code.

An alternative to platform-based isolation is language-based isolation [17, 32, 37, 48]. Language-based solutions, such as Caja [33], require the app developer to fetch all third-party JavaScript and rewrite it to satisfy the developer's security policy before serving it to the client. In contrast to language-based approaches, POWERGATE works with legacy apps and lets unmodified third-party JavaScript execute directly on the client machine. This reduces the burden on the app since the app's server does not need to fetch, rewrite, and serve all ads shown to the users. Note that deploying language-based isolation on the platform (as opposed to the server) would have been challenging because different apps have different isolation policies for native objects, while the platform presumably has a single JavaScript rewriting engine.

Vulnerable Firefox OS applications. DeFreez et al. [12] used lightweight static analysis to automatically find XSS (cross-site scripting) and other vulnerabilities in Firefox OS applications. Their focus and techniques are complementary to this paper, which investigates access control for native objects exposed to Firefox OS applications and similar applications on other platforms. We do analyze the exposure of native objects to untrusted Web origins, but the vulnerabilities we found are related to the unintended delegation of access rights, not XSS bugs in individual applications.

Permission re-delegation on Android. Many Android applications suffer from permission re-delegation vulnerabilities [11, 14]. This is an instance of the "confused deputy" problem [24], caused by a privileged application exposing sensitive operations without properly checking the credentials of the invoking application and thus enabling the latter to bypass the permission system.

While vulnerabilities analyzed in this paper are also of the general "confused deputy" type, they involve exposure of sensitive native APIs at the OS level, not at the application level. Furthermore, they are unrelated to Android permissions and affect platforms other than Android. Consequently, previously proposed defenses against re-delegation [6, 13, 14, 23, 30] do not solve the problems investigated in this paper.

Sandboxing for protecting OS resources. Many research projects [18, 19, 21, 25, 38, 45] and commercial products [35, 46] provide solutions for restricting an untrusted application to a subset of system resources. Unlike POWERGATE, these approaches do not differentiate between content from different Web origins as long as it is running inside the same application.

8 Conclusions

We analyzed the access-control models of the existing platforms for Web-based system applications and demonstrated semantic inconsistencies between the platforms, as well as design flaws that result in security vulnerabilities such as unauthorized access to OAuth tokens from untrusted Web code.

We then designed and implemented POWERGATE, a new access-control mechanism for Web-based system applications that allows developers to specify access policies for native objects in terms of high-level principals. POWERGATE policies are platform-independent and enforced uniformly and securely on all platforms. POWERGATE imposes a tiny performance overhead and is backward-compatible with all existing Web-based system applications.

Acknowledgments. This work was partially supported by the NSF grants CNS-0746888 and CNS-1223396, a Google research award, NIH grant R01 LM011028-01 from the National Library of Medicine, and Google PhD Fellowship to Suman Jana.

References

[1] Abusing WebView JavaScript bridges. http://50.56.33.56/blog/?p=314.

[2] D. Akhawe, P. Saxena, and D. Song. Privilege separation in HTML5 applications. In *USENIX Security*, 2012.

[3] AppArmor. http://developer.ubuntu.com/publish/apps/security-policy-for-click-packages/, 2014.

[4] A. Barth. The Web origin concept. http://tools.ietf.org/html/rfc6454.

[5] A. Barth, C. Jackson, and J. Mitchell. Securing frame communication in browsers. In *USENIX Security*, 2009.

[6] E. Chin, A. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys*, 2011.

[7] E. Chin and D. Wagner. Bifocals: Analyzing WebView vulnerabilities in Android applications. In *WISA*, 2013.

[8] Chrome app samples. https://github.com/GoogleChrome/chrome-app-samples, 2014.

[9] Permissions in Chrome apps and extensions. https://developer.chrome.com/apps/declare_permissions, 2014.

[10] Cordova platform support. http://cordova.apache.org/docs/en/3.4.0/guide_support_index.md.html, 2014.

[11] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *ISC*, 2010.

[12] D. DeFreez, B. Shastry, H. Chen, and J. Seifert. A first look at Firefox OS security. In *MoST*, 2014.

[13] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. Wallach. QUIRE: Lightweight provenance for smart phone operating systems. In *USENIX Security*, 2011.

[14] A. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security*, 2011.

[15] Firefox OS app permissions. https://developer.mozilla.org/en-US/Apps/Build/App_permissions, 2014.

[16] Firefox OS security model. https://developer.mozilla.org/en-US/Firefox_OS/Security/Security_model, 2014.

[17] M. Finifter, J. Weinberger, and A. Barth. Preventing capability leaks in secure JavaScript subsets. In *NDSS*, 2010.

[18] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX ATC*, 2008.

[19] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2004.

[20] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid Web/mobile application frameworks. In *NDSS*, 2014.

[21] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *USENIX Security*, 1996.

[22] M. Grace, W. Zhou, X. Jiang, and A. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WiSec*, 2012.

[23] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *NDSS*, 2012.

[24] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 1988.

[25] S. Jana, D. Porter, and V. Shmatikov. TxBox: Building secure, efficient sandboxes with system transactions. In *S&P*, 2011.

[26] X. Jin, X. Hu, K. Ying, W. Du, Y. Heng, and G. Peri. Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation. In *CCS*, 2014.

[27] X. Jin, T. Luo, D. Tsui, and W. Du. Code injection attacks on HTML5-based mobile apps. In *MoST*, 2014.

[28] K. Lin, D. Chu, J. Mickens, L. Zhuang, F. Zhao, and J. Qiu. Gibraltar: Exposing hardware devices to Web pages using AJAX. In *WebApps*, 2012.

[29] M. Louw, K. Ganesh, and V. Venkatakrishnan. AdJail: Practical enforcement of confidentiality and integrity policies on Web advertisements. In *USENIX Security*, 2010.

[30] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting Android apps for component hijacking vulnerabilities. In *CCS*, 2012.

[31] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on WebView in the Android system. In *ACSAC*, 2011.

[32] S. Maffeis and A. Taly. Language-based isolation of untrusted javascript. In *CSF*, 2009.

[33] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. http://google-caja.googlecode.com, 2008.

[34] WebView addJavascriptInterface remote code execution. https://labs.mwrinfosecurity.com/blog/2013/09/24/webview-addjavascriptinterface-remote-code-execution/.

[35] NSA. Security-enhanced linux. http://www.nsa.gov/research/selinux/.

[36] Ubuntu OnlineAccounts API. http://developer.ubuntu.com/api/html5/sdk-14.04/OnlineAccounts.OnlineAccounts/, 2014.

[37] J. Politz, S. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security*, 2011.

[38] N. Provos. Improving host security with system call policies. In *USENIX Security*, 2003.

[39] E. Shapira. Analyzing an Android WebView exploit. http://blogs.avg.com/mobile/analyzing-android-webview-exploit/.

[40] S. Shekhar, M. Dietz, and D. Wallach. AdSplit: Separating smartphone advertising from applications. *USENIX Security*, 2012.

[41] K. Singh. Practical context-aware permission control for hybrid mobile applications. In *RAID*, 2013.

[42] K. Singh, A. Moshchuk, H. Wang, and W. Lee. On the incoherencies in Web browser access control policies. In *S&P*, 2010.

[43] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postMessage in HTML5 websites. In *NDSS*, 2013.

[44] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in Android ad libraries. In *MoST*, 2012.

[45] W. Sun, Z. Liang, V. Venkatakrishnan, and R. Sekar. One-way isolation: An effective approach for realizing safe execution environments. In *NDSS*, 2005.

[46] SUSE. AppArmor Linux application security. https://www.suse.com/support/security/apparmor/.

[47] System applications working group charter. http://www.w3.org/2012/09/sysapps-wg-charter, 2014.

[48] A. Taly, U. Erlingsson, J. Mitchell, M. Miller, and J. Nagra. Automated analysis of security-critical JavaScript APIs. In *S&P*, 2011.

[49] Ubuntu Unity Web API. http://developer.ubuntu.com/api/devel/ubuntu-13.10/javascript/web-docs/, 2014.

[50] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *CCS*, 2013.

[51] App capability declarations (Windows Runtime apps). http://msdn.microsoft.com/en-us/library/windows/apps/hh464936.aspx, 2014.

[52] M. Zalewski. Browser security handbook. https://code.google.com/p/browsersec/wiki/Main.