Permission Re-Delegation: Attacks and Defenses

Adrienne Porter Felt
apf@cs.berkeley.edu
University of California, Berkeley

Helen J. Wang, Alexander Moshchuk
{helenw, alexmos}@microsoft.com
Microsoft Research

Steven Hanna, Erika Chin
{sch, emc}@cs.berkeley.edu
University of California, Berkeley

Abstract

Modern browsers and smartphone operating systems treat applications as mutually untrusting, potentially malicious principals. Applications are (1) isolated except for explicit IPC or inter-application communication channels and (2) unprivileged by default, requiring user permission for additional privileges. Although inter-application communication supports useful collaboration, it also introduces the risk of *permission re-delegation*. Permission re-delegation occurs when an application with permissions performs a privileged task for an application without permissions. This undermines the requirement that the user approve each application's access to privileged devices and data. We discuss permission re-delegation and demonstrate its risk by launching real-world attacks on Android system applications; several of the vulnerabilities have been confirmed as bugs.

We discuss possible ways to address permission re-delegation and present IPC Inspection, a new OS mechanism for defending against permission re-delegation. IPC Inspection prevents opportunities for permission re-delegation by reducing an application's permissions after it receives communication from a less privileged application. We have implemented IPC Inspection for a browser and Android, and we show that it prevents the attacks we found in the Android system applications.

1 Introduction

Traditional multi-user operating systems like Windows and Linux associate privileges with user accounts. When a user installs an application, the application runs in the name of the user and inherits the user's ability to access system resources (e.g., the camera). Browsers and smartphone operating systems, however, have shifted to a fundamentally new model where applications are treated as potentially malicious and mutually distrusting. Principals receive few privileges by default and are isolated from one another except for communication through explicit IPC channels. Only the user can grant individual applications permission to use devices and access user-private data (e.g., location) through system APIs. Consequently, each application has its own set of permissions, as granted by the user.

IPC in a system with per-application permissions leads to the threat of *permission re-delegation*. Permission re-delegation occurs when an application with a user-controlled permission makes an API call on behalf of a less privileged application without user involvement. The privileged application is referred to as a *deputy*, wielding authority on behalf of the user. The permission system should prevent applications from accessing privileged system APIs without user consent, but permission re-delegation circumvents this rule. This violates the user's expectation of safety when interacting with unprivileged applications. Permission re-delegation is a special case of the confused deputy problem [23] where authority is given by the user's permission.

We demonstrate that permission re-delegation is a realistic threat with a case study on Android applications. Android features per-application permissions and IPC, which are the necessary conditions for permission re-delegation vulnerabilities in applications. More than a third of the 872 surveyed Android applications request permissions for sensitive resources and also expose public interfaces; they are therefore at risk of facilitating permission re-delegation. We find 15 permission re-delegation vulnerabilities in 5 core system applications.

The threat of permission re-delegation is particularly important for web browsers, which are just beginning to add APIs that provide websites with access to devices and geolocation [32]. Additionally, an IPC primitive named `postMessage` has been widely deployed over the past few years, facilitating interaction between applications. Although device access for web applications is not yet widespread in 2011, permission re-delegation attacks will be a concern for future web applications. Addressing the problem of permission re-delegation prior to the full adoption of device APIs will be beneficial to future browser security.

We consider possible defenses against permission re-delegation attacks and propose IPC Inspection, a new OS mechanism that reduces a deputy's privileges after receiving communication from a less privileged application. Privilege reduction reflects that a deputy is under the influence of another application. Consequently, the permission system can deny a privileged API call from the deputy if any application in the chain of influence lacks the appropriate permission(s). We implement IPC Inspection for two different platforms: the Android operating system and ServiceOS's browser runtime [38]. Our Android implementation prevents all of the attacks we discovered in our case study. We evaluate the impact of IPC Inspection on applications and anticipate that most applications would require few changes to work with IPC Inspection, but some might require more permissions.

Contributions. We demonstrate that permission re-delegation is a widespread threat in modern platforms with per-application permissions and IPC. We also propose IPC Inspection, a new OS mechanism to defend against permission re-delegation.

Outline. We define the problem of permission re-delegation in Section 2. We then cast the problem in the context of today's web and smartphone platforms in Section 3. We describe our experience of discovering permission re-delegation vulnerabilities in Android in Section 4. We discuss possible defenses utilizing known techniques in Section 5 and then propose a detailed design for IPC Inspection in Section 6. We describe our implementation experience on Android and ServiceOS in Section 7. In Section 8, we evaluate the effectiveness of IPC Inspection. We discuss related work in Section 9.

2 Permission Re-Delegation

Permission systems prevent applications from performing actions that are not desired by the user. We are concerned with attacks on *user-controlled resources*, which are the resources guarded by permissions that are granted by the user. Devices like the camera and GPS are user-controlled resources, as are private data stores like lists of calendars and contacts. We do not consider attacks on resources not controlled by user-granted permissions, like memory or application-specific databases.

Permission re-delegation occurs when an application with a permission performs a privileged task on behalf of an application without that permission. This is a confused deputy attack [23] or privilege escalation attack. In this scenario, the user delegates authority to the *deputy* by granting it a permission. The deputy defines a public interface that exposes some of its functionality. A malicious *requester* application lacks the permission that the deputy has. The requester invokes the deputy's interface, causing the deputy to issue a system API call. The system will approve and execute the deputy's API call because the deputy has the required permission. The requester has succeeded in causing the execution of an API call that it could not have directly invoked (Figure 1).

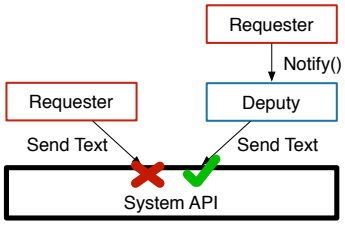

Figure 1: Permission re-delegation attack. The requester does not have the text message permission, but the deputy does. The deputy also defines a public interface with a `Notify` method, which makes an API call requesting to send a text message. When the requester calls the deputy's `Notify` method, the system API will send the text message because the deputy has the necessary permissions. Consequently, the attack succeeds.

Permission re-delegation can occur in three ways. First, an application may accidentally expose internal functionality to other applications. Second, a "confused" deputy might intentionally expose functionality, but an attacker might invoke it in a surprising context [23]. Third, the developer might expose functionality with the goal of attenuating authority but implement the attenuation policy incorrectly or in a way that is inconsistent with the system policy.

We cannot expect the deputy's developer to "opt in" to extra security measures or implement system policies. The deputy is neither helpful nor harmful to system security. Most developers are not security experts, and they are not independently motivated to prevent permission re-delegation because the consequences of permission re-delegation do not affect the deputy itself.

Although the deputy is trusted with some permissions, it is not trusted with all permissions. An application is trusted with precisely the set of permissions approved by the user, and the deputy and requester may have disjoint sets of dangerous permissions. A prevention mechanism cannot grant a deputy access to its requester's permissions unless the deputy already has the permissions.

We aim to equip the permission system with the ability to deny API requests made by a deputy on behalf of an unprivileged requester. Such an access control mechanism prevents the requester from executing privileged actions with side-effects or requesting sensitive data through another application. However, we do not address the problem of preventing a privileged application from sharing sensitive data that it has legitimately and independently obtained. We focus on protecting *access integrity*, whereas improper data sharing is a *confidentiality* problem. Preventing data leakage is a complementary problem beyond the scope of our work.

3 Scenarios

We discuss permission re-delegation as it applies to web browsers and smartphone operating systems.

3.1 Web Applications

Websites are mutually distrusting principals [37, 5, 3]. Today's browsers isolate websites from one another under the Same Origin Policy, which states that code from one site cannot access another site's content [33]. Traditionally, websites have also been prevented from accessing the user's local resources. However, this is changing as web applications begin to exhibit rich functionality.

Browsers are starting to offer APIs for accessing users' local resources. For example, the HTML5 device element [24] provides web applications with access to streaming audio and video data. The W3C Device APIs and Policy Working Group [35] is designing interfaces for contacts, calendars, messaging, cameras, and more. New versions of Firefox, Google Chrome, and Safari support preliminary versions of the HTML5 geolocation API [32]. Access to these APIs will be controlled by permissions that users grant to website origins.

Web permission re-delegation can occur in two ways:

1. *New windows.* In this scenario, a user unknowingly navigates to a malicious website. The malicious website opens a website with a permission in a new window. If the deputy website makes a privileged API call upon loading, then the malicious website has successfully mounted a permission re-delegation attack. This attack can be completely invisible to the user if the malicious requester loads the deputy in an invisible child frame; alternately, it can be hidden by opening the deputy as a pop-under window beneath the active browser window.

2. *Messages.* As in the previous scenario, a user unknowingly navigates to a malicious website. The malicious website sends a message to a deputy website that offers services to other websites via `postMessage`, an asynchronous client-side message passing channel. Requesters can send messages to new child frames or existing windows that are navigationally connected to the requester [25].

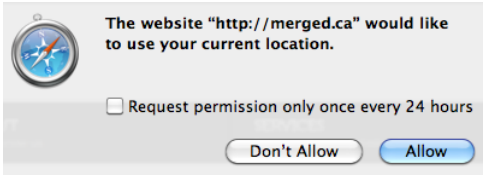

Figure 2: Safari 5 requests the user's permission before granting a website access to geolocation.

For example, consider a user who permanently grants a website the ability to record live video of the user for the purpose of streaming the video to a known web address (like Qik). The video website automatically begins simultaneously recording and streaming as soon as it is loaded. Later, the user visits a malicious website that loads the video website in an invisible child frame; as a result, the browser begins recording streaming live video without the user's knowledge.

Major browser vendors have recognized the problem of permission re-delegation. Mozilla Firefox, Safari, and Google Chrome implemented the geolocation permission with restrictions on child iframes. When a website is opened as a child iframe, it has no geolocation permission; the user is prompted for approval, even if the user has previously granted the website the geolocation permission. This prevents permission re-delegation attacks using iframes, but does not prevent permission re-delegation attacks on top-level windows or attacks between two iframes embedded in the same page. We also want to extend the defense mechanism beyond geolocation to all future permission-controlled browser APIs.

3.2 Smartphone Applications

Smartphone platforms like iOS, Android, and Windows Phone 7 support third-party application markets. The markets have a low cost of entry, and not all of the developers are equally trustworthy. Consequently, smartphone operating systems treat applications as potentially malicious. Smartphone operating systems also provide APIs to phone devices (Bluetooth, camera, GPS, etc.) and the network. Upon user approval, smartphone operating systems grant per-application access to these resources.

This paper focuses on Android, although our work applies to other smartphone operating systems with user-controlled application permissions and IPC. Android permissions are categorized into 3 security levels: Normal, Dangerous, and Signature.[1] Normal permissions protect API calls that could annoy but not harm the user (e.g., `SET_WALLPAPER`); these do not require user approval. Dangerous permissions let an application perform harmful actions (e.g., `RECORD_AUDIO`). Signature permissions regulate access to extremely dangerous privileges, e.g., `CLEAR_APP_USER_DATA`. Malware with Dangerous or Signature permissions can spy on users, delete data, and abuse their billing accounts [7, 27, 34].

Android applications may communicate with each other, which introduces opportunities for permission re-delegation. Inter-process messages known as *Intents* are used for communication. Applications can make four types of components publicly available:

[1] We group SignatureOrSystem and Signature permissions together since there is no significant distinction between the two categories.

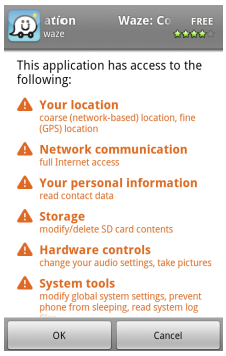

Figure 3: Android displays permissions for user approval during the installation of an application from the Android Market.

- *Services* run in the background. They can be started with Intents or "bound" for synchronous calls.
- *Activities* provide applications with user interfaces. They can be started with Intents. An Activity can optionally provide a return value that is delivered when the user finishes interacting with the Activity.
- *BroadcastReceivers* handle broadcast Intents. They run in the background. Sometimes they invoke a Service or an Activity to handle the task.
- *ContentProviders* are local databases.

Services and BroadcastReceivers are inviting targets for stealthy permission re-delegation attacks because they may not have a visible user interface. A developer can restrict access to a component by specifying in the manifest that requesters must have a given permission or dynamically checking the caller's identity at runtime.

3.3 Granting Permissions

Current permission systems ask users to grant permissions in one of two ways:

Time-of-Use. In a time-of-use permission system, users are prompted to approve or deny permission for a resource when a privileged API call is made. Web browsers and Apple iOS use this type of permission for third-party applications. The permission may be granted permanently, for a period of time, or for a single use. Figure 2 provides an illustration of Safari 5 asking a user to grant a time-of-use permission.

Install-Time. In a system with install-time permissions, an application declares its permission requirements in a manifest file. The user is prompted to approve the permissions during the installation process, and the application is only installed if the user grants permanent permissions to the requested resources. In a strict install-time permission system, new permissions cannot be requested during runtime. Android and Windows Phone 7 use install-time permissions. Figure 3 shows an example of an Android installation permission prompt.

	Public Service	Public Receiver
Dangerous permissions	84	269
Signature permissions	13	34

Figure 4: We surveyed 872 Android applications and identified ones with both public components and notable permissions.

4 Case Study: Attacks on Android

We perform a case study on Android applications to demonstrate that permission re-delegation is a real-world concern. We find that many applications are at risk of containing a vulnerability, and we build example attacks using core system applications.

4.1 At-Risk Applications

An application is *at risk* of containing a permission re-delegation vulnerability if it both requests permissions and exposes a public interface. In particular, we are interested in stealthy attacks that can be conducted in the background. We examine a set of 872 applications, which is composed of the 16 core system applications that come pre-installed with Android 2.2, the 756 most popular free applications from the Android Market, and the 100 most popular paid Market applications.

For each application, we parse its manifest to find its list of permissions, public Services, and public Receivers. Services always run in the background, and Receivers might run in the background. By evaluating whether an application has both permissions and a Service/Receiver, we can identify at-risk applications. We discard public components that are protected by permissions in the manifest, as they may not be at risk.

It is also possible for an application to perform a dynamic check on its caller's permissions, which would not be reflected in our manifest analysis. 9% of all applications in our set perform dynamic permission checks; however, the permission checks are not often used to prevent external use of Services or Receivers. (They are typically used to protect Providers, which we do not consider, or by embedded advertising libraries to determine what operations they can perform.) We examined a set of 50 randomly selected applications with public components and found that only 1 application does so to protect a Receiver or Service.

Figure 4 shows the highest-level permission of an application, and whether it also has an unprotected public component. Overall, 320 of the 872 applications (37%) have permissions and at least one type of public component. 11 of the 16 system applications are at risk.

4.2 Vulnerabilities

We examine the at-risk system applications to identify stealthy permission re-delegation vulnerabilities. An application contains a vulnerability if there exists an exploitable path in the application between a public entry point and a restricted system API call. We construct attacks using 15 vulnerabilities in 5 applications. These vulnerabilities are present on every Android 2.2 phone.

Finding Attacks. To find vulnerabilities, we created a path-finding tool. We first disassemble the at-risk system applications using Dedexer [31]. Our tool then parses the disassembled applications to find method declarations and invocations; from the results, we build the call graphs of the applications. Searching the call graphs reveals paths from public entry points to protected system API calls. This tool likely misses viable attack paths; our call graph analysis does not handle inheritance relationships or detect flow through structures like callbacks. We also were only able to look for attacks on a subset of the API due to incomplete documentation.

We identify valid attacks by building test cases for the paths produced by our path-finding tool. We only consider attacks on API calls with verifiable side effects, so that we can tell if an attack succeeds. We are also only able to look for attacks on a subset of the API because Android permission documentation is incomplete.

Results. We built attacks using 5 of the 16 system applications. All of the attacks succeed in the background with no visible indication to the user. These 5 applications provide 15 paths to 13 interfaces with permissions, one of which is SignatureOrSystem and nine of which are Dangerous. We present two example permission re-delegation attacks:

Settings serves as the phone's primary control panel. When a user presses certain buttons, Settings' user interface sends a message to a Settings BroadcastReceiver that turns WiFi, Bluetooth, and GPS location tracking on or off. However, Settings' BroadcastReceiver accepts Intents from any application. An unprivileged application can therefore ask the Settings application to toggle the state of devices by sending its BroadcastReceiver the same Intents that it expects from its user interface. Turning these devices on or off is supposed to require Dangerous permissions (`CHANGE_WIFI_STATE`, `BLUETOOTH_ADMIN`, and `ACCESS_FINE_LOCATION`, respectively).

Desk Clock provides time and alarm functionality. One of its public Services accepts directions for playing alarms. If an unprivileged application sends an Intent requesting an alarm with no end time, then Desk Clock will indefinitely vibrate the phone, play an alarm, and prevent the phone from sleeping. The alarm will continue until the user kills the Desk Clock process. Playing sound with a wake lock requires the Dangerous `WAKE_LOCK` permission, and vibrating the phone requires the Normal `VIBRATE` permission.

We found concrete vulnerabilities in 5 of the 16 applications, which amounts to half of the 11 system applications that we identified as at-risk. It is likely that the other at-risk system applications also contain vulnerabilities that we did not uncover. (Our call graph tool is limited, as discussed above.) We have notified the Android team; several of the vulnerabilities have been confirmed and fixed [16, 15, 17, 18].

5 Defense Discussion

We present our requirements for a permission re-delegation defense mechanism and consider whether existing techniques can satisfy these goals.

5.1 Goals

Our goals for a successful permission re-delegation defense mechanism are:

1. *Preventing permission re-delegation*: Applications should not be able to re-delegate permissions for user-controlled resources.
2. *Runtime independence*: We want the solution to be language- and runtime-independent. This is advantageous because many platforms support applications that are written in different programming languages and run on different runtimes.
3. *Developer independence*: Given evidence that developers are unmotivated to proactively prevent permission re-delegation (Section 4.2), a solution cannot rely on developer diligence for security.
4. *Ease of development*: The mechanism should not impose an excessive burden on developers; applications should retain their functionality.
5. *Dynamic*: The defense mechanism must work at runtime and not depend on application analysis. Client-side application analysis is not feasible for web applications because website code can change and encompass arbitrarily many documents.

Our focus is on controlling access to resources; it is not our goal to protect data privacy. We wish to prevent a privileged piece of data from being accessed, but we do not aim to prevent it from being shared once it has been accessed. Protecting data privacy can be achieved with complementary solutions like TaintDroid [12].

5.2 Potential Defenses

Capabilities. A *capability* is an unforgeable, shareable token that, when used, grants access to a privilege [23]. To prevent permission re-delegation, a deputy could ask its requester to provide a capability and use it to make system API calls. However, this approach does not meet our goal of developer independence; a poorly written deputy could use its own capabilities rather than its requester's when making system API calls.

Alternately, a system could control access to privileges by requiring approval before granting an application the ability to communicate with a deputy. This would require static analysis of the deputies installed on the client to understand what user-relevant privileges the communication would involve. Following the example in Figure 1, an application that wants to use the "Notify" deputy would need to be authorized to use the underlying "Send Text" authority. Static capability provisioning is a topic for further exploration, but it can only be applied to platforms where static analysis of deputies is possible.

Taint Tracking. In a taint tracking-based solution, the requester's data could be the source of taint, and the taint could propagate as the requester's data interacts with the deputy. If a deputy makes a privileged API call and the call is tainted, then the system could become aware that permission re-delegation has occurred. Unfortunately, tracking both data flow and control flow in a runtime-independent manner incurs more than an order of magnitude of performance overhead [14]. Furthermore, taint tracking both data and control flow would likely lead to taint explosion. Taint explosion would make the system unnecessarily restrictive. A taint tracking-based solution would need to track both direct and indirect taint flows because not all permission re-delegation attacks are data-dependent. For example, not all API calls require parameters, and some applications process IPC calls without reading the actual message.

MAC. Mandatory access control (MAC) systems (e.g., [9, 20]) are centralized information flow control systems where the operating system enforces a fixed information flow control policy across integrity or confidentiality levels. Such systems mandate that no information can flow from low-integrity principals to high-integrity principals or from high-confidentiality to low-confidentiality principals. Our goal of preventing permission re-delegation can be cast as MAC to some extent: a permission set could be treated as an integrity label, and A would have a lower integrity level than B if A has a permission that B does not. We are then concerned about safe information flow with respect to access to user-controlled resources.

However, Android applications cannot be strictly ordered because applications often do not fit the subset relationship. When this happens, neither the deputy nor the requester has a strictly higher integrity level. This presents an application functionality problem: when a requester initiates communication with a deputy, the requester cannot receive the response if it has a permission that the deputy lacks. Since Android applications commonly have intersecting but non-subset permission sets, this represents a significant functionality problem. Section 8.2 describes examples of applications for which this would be prohibitively restrictive.

Stack Inspection. Stack inspection [19, 36] is used in Java Virtual Machines and the Common Language Runtime to prevent confused deputy attacks within a runtime. When a deputy makes a privileged API call, the system checks whether the call stack includes any unprivileged applications. Principals in the permission re-delegation threat model operate in separate runtimes; to adapt to this scenario, the runtime could annotate the bottom of the stack with the requester's identity when delivering a message event or starting the application.

Standard stack inspection has several shortcomings. First, the approach is dependent on the runtime for correctness and would need to be re-implemented repeatedly for a system with multiple types of runtimes. Second, stack inspection cannot prevent permission re-delegation when the deputy's API call is de-synchronized from the request because the requester does not appear on the call stack. For example, JavaScript is event-driven after the initial document loads, and each event has a different stack; and Android applications often make internal IPC calls (each of which resets the stack) in order to complete a single operation.

HBAC. History-based access control (HBAC) reduces the permissions of trusted code after any interaction with untrusted code [1]. Like stack inspection, HBAC relies on runtime mechanisms and does not achieve our goal of runtime independence. Like MAC, HBAC performs permission reduction upon receipt of return values, which places constraints on application functionality.

6 IPC Inspection

We build upon existing techniques to propose a new defense for permission re-delegation. We track information flow through inter-application messages, but not within an application. Our solution is similar to stack inspection or HBAC, but modified to address their limitations. Like HBAC, we perform privilege reduction following communication, but we apply our mechanism at the OS level rather than as part of a runtime.

6.1 Our Design

We propose *IPC Inspection*. When an application receives a message from another application, we reduce the privileges of the recipient to the intersection of the recipient's and requester's permissions. We consider an application to be acting as a deputy on behalf of a requester once it has received communication from the requester. The deputy's current set of permissions captures the communication history of the deputy and other applications. Privilege reduction does not remove privileges that are not controlled by the user.

IPC Inspection carries the same semantics as stack inspection, but we generalize method invocations to IPC calls and externalize intra-application asynchronies like message queues. IPC Inspection is also runtime- and language-independent.

Basic Rules. IPC Inspection is comprised of three primary mechanisms. First, we maintain a list of current permissions for each application. Second, we build privilege reduction into the system's inter-application communication mechanisms. Starting an application and sending an explicit message both count as IPC. Third, we allow a receiving application to accept or reject messages. Applications can limit who they receive messages from by registering a list of acceptable requesters. Depending on the platform, acceptable requesters can be identified individually (e.g., by domain) or based on their set of permissions (i.e., any application with permission Y). This prevents privilege reduction from being abused as a denial of service mechanism.

More precisely, we define four basic rules to govern access rights and privilege reduction. We write $A \rightarrow B$ to indicate that application A sends a message to application B, and let $P^t(A)$ denote the set of permissions held by application A at time t. The rules follow:

1. Initial state: $P^0(A) = P^{Original}(A)$. When an application starts running, it begins with the permissions that were granted by the user.
2. Privilege reduction for recipient: If $R \rightarrow D$ at time t, then $P^t(D) = P^{t-1}(D) \cap P^{t-1}(R)$. When an application receives a message, its permissions are reduced to the intersection of its and the sender's current permissions.
3. Sender's permissions remain unchanged: If $R \rightarrow D$ at time t, then $P^t(R) = P^{t-1}(R)$.

Several properties of privilege reduction follow from the access rights rules:

- Transitivity. An application's current permissions reflect the permissions of all of the applications in a chain of communication. If $R_1 \rightarrow R_2$ at time t, and $R_2 \rightarrow D$ at time $t+1$, then $P^{t+1}(D) = P^{t-1}(R_1) \cap P^{t-1}(R_2) \cap P^t(D)$.
- Additivity. If an application receives messages from multiple applications, then its permissions will be repeatedly reduced. $P^t(D) = P^0(D) \cap \bigcap_{i=1}^{t-1} P^i(R_i)$, where $R_i \rightarrow D$ for each time i.
- Bounds. An application's current permissions can never exceed its original permissions (i.e., $P^i(A) \subseteq P^0(A), \forall i$); there is no mechanism for increasing permissions.

Privilege reduction requires the platform to compute the intersection of two permissions, and this intersection function will differ by permission scheme. For example, permissions can be hierarchical, temporal, or monetary (as in $5 for text messages). When calculating the intersection, the lesser value needs to be taken: the permission lower in the hierarchy, the lower monetary limit, or the shorter temporal permission.

Non-Simplex IPC. The basic rules apply to *simplex* (unidirectional) communication. Simplex communication implies a clear relationship: the requester sends a message or starts an application, and the deputy acts. However, an operating system can offer other communication mechanisms. With *request-reply* IPC, the recipient of a message returns a value. For example, an Android Activity may return a result upon completion. The roles of deputy and requester remain clear with request-reply IPC, so IPC Inspection does not reduce the requester's permissions when it delivers a reply. In contrast, *duplex* communication is a stream of data that flows between two applications (e.g., TCP sockets). Both applications can act as a deputy or requester during duplex IPC, so IPC Inspection reduces the privileges of both. Applications can implement alternate protocols atop these three primitives, but the OS will not be aware of them.

IPC Inspection is similar in spirit to Mandatory Access Control, but IPC Inspection is less strict because it does not enforce privilege reduction in both directions after request-reply communication. This decision is in the interest of application functionality: highly-privileged requesters would be unable to accept responses from less-privileged deputies without risking loss of privilege. Although there is a chance that a permission re-delegation attack could stem from the receipt of a return value, it is not a common case. Section 8.2 describes examples of applications that would not be able to function if return values prompted privilege reduction.

HBAC also reduces privileges based on return values but attempts to preserve application functionality by providing authors with explicit rights restoration. In HBAC, an application can validate a return value and then request to have its removed privileges reinstated. Although rights restoration solves the functionality problem, it relies upon developers correctly and non-spuriously restoring their permissions. Developers could abuse this mech-

anism by restoring permissions in every permission failure exception handler. Therefore, we choose not to support explicit rights restoration in IPC Inspection.

Application Instances. Deputies may need to simultaneously interact with the user and multiple requesters. For example, the user might be interacting with an application when it receives a message from a less-privileged requester; the ensuing privilege reduction caused by the requester would interfere with the user's experience. To prevent privilege reduction from impeding application functionality, we create new application instances to handle messages. All applications have a primary instance, which is the instance of the application that the user interacts with. When a requester asks the system to send a message to the deputy, the system automatically starts a new instance of the application. Multiple instances of the same application will run concurrently, in their own isolation units with their own current permissions.

As a performance optimization for install-time permission systems, it is not always necessary to create a new instance. The primary instance can be used for requests that do not prompt privilege reduction. In an install-time permission system, we know that privilege reduction is not necessary if the deputy already lacks permissions or the requester has a superset of the deputy's permissions. Instance reuse is not possible with time-of-use permission systems because the deputy could dynamically request more permissions; it would not be clear which requester is responsible for the permission prompt.

Some applications cannot exist in duplicate. Long-running background processes may have state that cannot be multiply instantiated. For this purpose, the system can let applications request to be *singletons*. All communication events will be dispatched to the same instance for a singleton application, and a singleton application's permissions will be repeatedly reduced upon the delivery of each communication event until the application exits.

Our altered version of Android automatically creates a new instance for every communication event unless a deputy asks to be a singleton; our browser implementation creates a new instance whenever a requester programmatically opens a new window, but not when messages are sent to an existing window. The singleton design pattern does not make sense from a web application perspective because websites already expect to be simultaneously open in multiple windows in the same browser.

Circumvention. A malicious deputy could circumvent the IPC Inspection rules: a developer could place information about a request in storage and then perform the request later when the deputy regains full privileges. This does not violate any of our goals. We are not concerned with deputies maliciously sharing permissions; after all, a malicious deputy could directly abuse its permissions. Instead, we are concerned about benign deputies thoughtlessly or accidentally giving away privileges to other applications. We expect that the pattern of saving requests in storage would be rare in practice.

Permission re-delegation attacks could be mounted using return values from request-reply IPC. In this attack, a privileged application sends a message to an unprivileged application. The unprivileged application returns a malicious value, which causes the privileged application to perform a malicious action. We do not defend against this attack in the interest of application functionality. Our policy of disregarding return values is similar to the policy enforced by stack inspection.

6.2 Platform Proposals

We discuss the impact of IPC Inspection rules on permissions and communication, and we present proposals for how systems could add extra support for IPC Inspection.

Permission Requests. IPC Inspection changes how users and developers interact with permissions. The impact of IPC Inspection depends on whether the system uses time-of-use permissions or install-time permissions. Time-of-use permission systems are the simpler case: in a time-of-use permission system like the browser, the user could be prompted whenever permission re-delegation is detected. For example, "Allow R and D to send text messages?" If the user answers affirmatively, the API call completes. As such, time-of-use permission systems can accommodate IPC Inspection without changes to the developer experience.

The relationship between IPC Inspection and install-time permission systems (e.g., Android) is more complex. If IPC Inspection is used with a pure install-time permission system, then an application's developer must request all of the permissions used by the deputies that the application interacts with. First, this might be hard to determine. Second, this could lead to permission bloat: a requester must have all of the permissions needed by its deputy or deputies, even if the requester never individually uses the permissions. To prevent this, we propose the relaxation of install-time permission requirements when IPC Inspection is applied to a platform. If permission re-delegation is detected, the platform could prompt the user to grant the requester temporary access to the privilege via the deputy. If granted, the deputy's permission would be restored. This would prevent requester applications from needing to request permissions that they will not use independently from deputies. This change would require usability studies to determine whether it effects user understanding of permissions.

Request-Reply for the Web. Today's websites exchange messages using `postMessage`, a simplex communication primitive. Websites that wish to use request-reply semantics must construct the necessary support on top of `postMessage`; e.g., such support is built into the popular jQuery library. Unfortunately, the browser is unaware of such application-level semantics and must apply IPC Inspection rules to all `postMessage` recipients, which may conflate the requester and the deputy. When a requester receives a `postMessage` corresponding to a return value, the browser will treat it as a deputy and unnecessarily reduce its privileges.

Current web standards lack a request-reply IPC primitive that would let browsers avoid this problem. We propose adding such a primitive by extending `postMessage` to take an optional callback argument. Replies would not trigger privilege reduction.

Device Policies. Instance reuse is not possible for systems with time-of-use permissions, as discussed in Section 6.1. However, it would be possible if the browser could identify sites that do not ask for any permissions. We propose that browsers only grant device access to web applications if they statically declare that they will ask for permissions. Web sites without permissions would never need to be multiply instantiated. This could be implemented, for example, with Content Security Policies, which already support developer-authored restrictions on what scripts, images, etc. can be loaded into a page [28]; a new rule would enable device access.

7 Implementation

We implemented two IPC Inspection prototypes: one for Android and one for ServiceOS's browser runtime.

7.1 Android

We implemented a prototype of IPC Inspection as part of Android 2.2. We added support for IPC Inspection to the PackageManager and ActivityManager. The PackageManager installs applications, stores their permissions, and enforces permission requirements. The ActivityManager handles communication between applications and starts applications as necessary.

Five events trigger privilege reduction: starting a Service, binding a Service, starting an Activity, receiving a Broadcast Intent, and requesting a ContentProvider. Our altered ActivityManager notifies the PackageManager whenever any of these five communication events occurs. One notable exemption is the Launcher system application, which we allow to communicate with any other application freely, to prevent privilege reduction from occurring whenever the user launches applications.

When the PackageManager is notified of a pending communication event, it checks whether privilege reduction needs to occur. The message is dispatched normally, without privilege reduction, if (1) the message is from the system process, or (2) the requester has all of the target's permissions. Otherwise, privilege reduction of the target occurs before the message is delivered.

The mechanics of privilege reduction differ based on whether the target application is a singleton. The PackageManager reduces privileges of singleton applications by removing the appropriate permission(s) from the data structure that assigns permissions to application UIDs. An application can request to be a singleton by setting a `singleton` value in its manifest. For non-singleton applications, the PackageManager instructs the ActivityManager to create a new instance of the application to receive the message. The ActivityManager places each new instance in a new process, with the same UID as the application's primary instance. When the instance's process is created, the PackageManager records the removed permissions in a data structure associated with the instance's PID. Instances of an application have access to the same files because they share a UID. Android also uses UIDs as a security boundary between applications, but we assume instances are not trying to attack each other. For both singletons and non-singletons, the PackageManager records which requester is responsible for the removal of removed permissions in a `blame` map.

Permission enforcement in our modified version of Android occurs in two steps. First, the standard permission enforcement mechanism checks whether the given permission is assigned to the application's UID. This check will return the same result for all instances of an application, since permissions are associated with UIDs. If the standard permission check succeeds, then our altered PackageManager additionally checks whether the permission is in the process's list of removed permissions for that instance. If it is, then access is denied. The `blame` map allows the PackageManager to identify the requester that is responsible for a permission failure. Following our proposal in Section 6.2, the operating system could then ask the user to temporarily grant the permission, although we did not implement this user interface.

We also extend the manifest file format so that deputies can limit incoming messages. The existing Android `permission` attribute lets an application specify a permission that a requester must have. We extend this to accept a set of permissions. If the developer limits the receipt of incoming messages to requesters with adequate permissions, then the application will never need to undergo privilege reduction.

7.2 ServiceOS

To demonstrate IPC Inspection in the context of web applications, we implemented IPC Inspection as a permission manager for ServiceOS, a client platform that supports both web and desktop applications [38]. Our implementation enabled IPC Inspection for ServiceOS's browser runtime.

In ServiceOS, each web origin is a principal as defined by the Same Origin Policy [33]. Permissions for user-controlled resources are granted on a per-principal basis. When a user navigates to a website by following a link or entering a new URL into the location bar, the resulting window is associated with the appropriate site principal. The user is asked to grant or deny permissions for that origin. The permission manager keeps track of all current permissions and controls access to a mock device API that represents the new HTML5 APIs.

The browser's communication system informs the permission manager of communication events. Two events trigger privilege reduction:

PostMessages. When a website `R.com` sends a `postMessage` to a window belonging to `D.com`, this communication passes through the IPC mechanism in the browser. Consequently, `D.com`'s permissions are subject to reduction. Permissions are restored when all windows belonging to a principal are closed. To prevent DOS, we provide websites with the ability to limit which origins they receive `postMessages` from; messages from other origins will be dropped by ServiceOS.

New Windows. When a website `R.com` creates a new window (e.g., a child frame) belonging to `D.com`, we treat this as a new service request; the parent window is the requester and the new window is the deputy. We consequently create a new principal for the new window, isolated from the rest of its origin. The new window's privileges are immediately reduced. Unfortunately, we cannot provide a mechanism for limiting who a web site can be opened by; websites are typically opened by so many others that a whitelist is not realistic.

As with the Android implementation, we record privilege reduction so that a correct prompt could be displayed to the user to explain the permission failure. Additionally, this implementation could be re-implemented in any major browser.

	Action	Data
Normal	15	26
Dangerous	59	31
Signature/System	10	1
Total	84	58

Figure 5: 142 Android API calls, classified.

8 Evaluation

We evaluate IPC Inspection with respect to security and ease of application development.

8.1 Effectiveness

Our primary goal is to prevent permission re-delegation. We evaluate IPC Inspection for Android security.

Scope. IPC Inspection strengthens access control for user-controlled resources and prevents applications from making unauthorized API calls. Now, we evaluate the scope of protection on Android system APIs.

We consider 142 methods from the Android API that are protected with permissions and classify each method as *action* or *data* calls. Action calls have side effects, and data calls return values without side effects. The set of 142 methods includes all of the protected methods in the SDK documentation, plus additional protected methods we identified using randomized testing. There are likely more protected interfaces to be identified, but we believe that the set of 142 methods is a representative sample of the full set of protected interfaces. We classify each method according to its description in the documentation. Accessors are typically data calls and mutators are typically action calls.

IPC Inspection can prevent both action and data calls from being invoked when an application is acting under the influence of another, less-privileged application. Nevertheless, IPC Inspection does not provide privacy for the return results of data calls, if the API call was not made on behalf of the requester. For example, an application with privileged geolocation data may pass a cached location value to a less privileged application. We emphasize, however, that IPC Inspection does prevent an application from obtaining the data while under the influence of another application.

We conducted a measurement on the makeup of action and data on Android. Figure 5 shows the results. Nearly 70% of the interfaces protected by Dangerous and Signature/System permissions are action calls.

Attack Prevention. Our Android implementation prevents all of the permission re-delegation attacks described in Section 4.

We suspect that many Android applications do not truly intend for their publicly invokable interfaces to be public; instead, the interfaces are intended for internal communication or messages from the operating system. Some messages that are typically sent by the operating system can also be sent by non-system applications; if the application does not additionally check the identity of its caller, it can be confused into performing an action. For example, we found in Section 4 that the Phone appli-

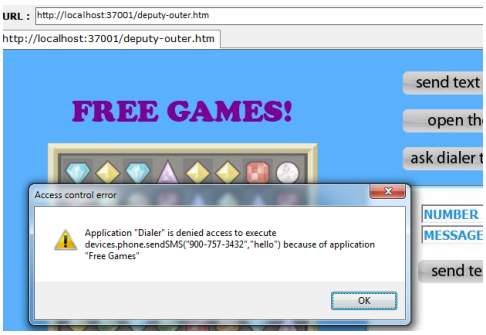

Figure 6: In this ServiceOS test attack, a "Game" application with no permissions opens a "Dialer" application from a different domain as a child frame. The user previously granted the Dialer the ability to send text messages, but the permission is removed upon loading because the Game lacks it. The Game sends the Dialer a `postMessage` asking the Dialer to send a text message. The Dialer's API call is denied.

cation will mute indications of an incoming phone call based on a message it expects to receive from the system. Tests on 5 applications that appear to fall in this category indicate that IPC Inspection prevents unintentionally public interfaces from being surprisingly invoked.

We built attack test suites for both Android and ServiceOS, and our implementation prevents all of the test attacks from succeeding. Each test suite is comprised of a set of communications from an unprivileged application to a privileged application. The privileged application is set up to make an API call following the receipt of any type of message. In a successful test, IPC Inspection prevents the API call from completing. We exercise all of the communication events available in each platform. Figure 6 is a screenshot of a test attack in ServiceOS.

8.2 Ease of Development on Android

Although we aim to prevent permission re-delegation, we do not want to prevent legitimate application interactions. We discuss four categories of applications: non-deputies, intentional deputies, unintentional deputies, and requesters. An application is an *intentional* deputy if it is built to expose functionality to other applications, whereas an application is an *unintentional* deputy if it exposes internal functionality accidentally. The Android communication system makes it easy for applications to accidentally expose internal functionality [6]. We estimate the prevalence of these four types of applications.

Non-Deputies. An application that does not offer services to other applications will not be greatly impacted by IPC Inspection. A non-deputy application will not need to be multiply instantiated, nor will it experience privilege reduction.

Unintentional Deputies. IPC Inspection will prevent malicious applications from using accidentally public interfaces of unintentional deputies to launch permission re-delegation attacks. The application will not be affected during normal operation.

Intentional Deputies. Applications that do provide public services can take one of two approaches, depending on their needs. The first option is that a deputy can accept calls from arbitrary requesters. A developer that chooses this option should place security exception handling code around API calls that require permissions, in case an unprivileged requester causes privilege reduction of an instance. The second option is for an application to require that potential requesters have all of the permissions necessary to make the relevant API calls. A developer that chooses this option therefore needs to specify a list of required permissions. This choice obviates the need for multiple instances, which is beneficial from a performance perspective. However, it may reduce the number of eligible requesters. A singleton application should choose this option to prevent its primary (and only) instance from experiencing privilege reduction.

Here, we discuss the impact of IPC Inspection on three popular, real-world intentional deputies:

Barcode Scanner. The "ZXing" barcode scanner is among the 50 most popular free applications in the Android Market. It provides public interfaces for scanning, creating, and displaying barcodes. ZXing uses several permissions to complete these tasks (e.g., `CAMERA`). ZXing correctly attenuates authority by asking for user permission before performing privileged tasks, so IPC Inspection does not add security in this case. ZXing can be repeatedly instantiated without any apparent issues, so ZXing does not necessarily need to limit its requesters to applications with certain permissions.

E-Mail. "GMail" is the official Google e-mail client. It provides several public interfaces. The primary public interface is an e-mail composition Activity that can be pre-seeded by the requester. GMail uses several permissions to send a pre-composed e-mail, e.g., `WRITE_EXTERNAL_STORAGE` (for uploading file attachments) and `INTERNET`. It is not clear whether all of GMail's other public interfaces are truly intended to be public: for example, one BroadcastReceiver listens for a message that indicates login accounts have changed. Like ZXing, GMail can be repeatedly instantiated without exhibiting obvious flaws.

Music Player. "Music" is one of the pre-installed system applications. It provides many public interfaces, including a MediaPlaybackService. The MediaPlaybackService opens music files, starts and stops music playback, and manages the current playlist. It uses permissions

such as WRITE_EXTERNAL_STORAGE (to open files) and WAKE_LOCK (to keep the phone on while playing music). We discovered in Section 4 that permission re-delegation attacks can be mounted using the MediaPlayBackService, but they are prevented with our Android IPC Inspection implementation. MediaPlayBackService needs to run as a singleton because it is a long-running background service that maintains state.

In summary, ZXing and GMail developers can choose whether to write exception handling code or lists of permission requirements for their requesters. The Music application is a singleton, so its developer should accept requests only from applications with all of its required permissions. The developer must specify that Music is a singleton and list the desired requester permissions.

Requesters. Under IPC Inspection, deputies may require their requesters to have more permissions. We present three example requesters that make use of the deputies presented above and consider whether they already have the necessary permissions. Additional install-time permissions would not be necessary if Android were to allow time-of-use permissions for the specific case of interacting with deputies.

We also consider the effects of a hypothetical rule that reduces privileges for requesters upon receipt of a reply value. Stricter policies (MAC and HBAC) include such a rule, as discussed in Section 6.

Barcode Scanner. Many applications rely on the ZXing barcode scanner [41]. One example is "Beer Cloud," which lets users find nearby bars that serve particular beers. Beer Cloud invokes the ZXing barcode scanner to identify beers. Under IPC Inspection, Beer Cloud would require the CAMERA permission to interact with ZXing. Currently, Beer Cloud does not have the CAMERA permission; IPC Inspection would require it to add it, which might overprivilege the Beer Cloud application.

Once Beer Cloud has received the barcode data from ZXing, it passes the beer information and user location to a backend server. The server returns nearby bar addresses. Beer Cloud uses Internet and location permissions to accomplish this. If we were to implement privilege reduction following return values, then Beer Cloud would not be able to pass the beer and location data to its backend server because ZXing does not have the necessary location permission.

E-Mail. "Blackmoon File Browser" relies on GMail for file sharing. Blackmoon File Browser only has one permission, WRITE_EXTERNAL_STORAGE. Under IPC Inspection, it would require the INTERNET permission as well. None of GMail's public interfaces return values, so a hypothetical return value rule would not impact Blackmoon File Browser or any other requesters of GMail.

Intentional Deputy	5 applications
Unintentional Deputy	4 applications
Requester	6 applications

Figure 7: We classify 20 Android applications. 13 applications are deputies or requesters. One is both an intentional and unintentional deputy, and another acts as both a deputy and a requester. All have Dangerous permissions.

Music Player. "ScrobbleDroid" uses the Music application's MediaPlayBackService to track the user's recently played songs. The recently played songs are then posted on the website last.fm. Binding to the singleton MediaPlayBackService would require ScrobbleDroid to add four permissions under IPC Inspection. MediaPlayBackService does not actually use all four of the extra permissions (they are used elsewhere in Music), so this would slightly over-privilege ScrobbleDroid. In the reverse direction, ScrobbleDroid uses return values provided by the MediaPlayBackService. Since ScrobbleDroid has a permission that Music does not have, ScrobbleDroid would be impacted by the hypothetical return result rule that we rejected in Section 6.

In summary, Beer Cloud and Blackmoon File Browser would need to gain user approval for additional permissions that make sense considering their functionality. However, they wouldn't use the permissions for anything but communication with deputies. ScrobbleDroid would need otherwise unnecessary permissions because Music is a singleton. Beer Cloud and ScrobbleDroid also illustrate why we do not reduce privileges for request-reply message exchanges.

Prevalence. We consider 20 randomly selected Android applications (from our set of 872) and evaluate whether they act as deputies, requesters, or both. We manually interact with the applications' user interfaces, log communication events, and examine their manifests.

Figure 7 shows the results of the survey. Under IPC Inspection, developers of intentional deputies and requesters may need to make minor changes to their applications: intentional deputies might need to specify permission requirements for requesters, and requesters might need to add extra permissions. 11 of the 20 applications are intentional deputies, requesters, or both.

We also classify 4 of the 20 applications as unintentional deputies. IPC Inspection would prevent these accidentally public interfaces from being used for permission re-delegation attacks. The developer of the first unintentional deputy obviously copied part of the manifest from another application with public interfaces. The second unintentional deputy's public interface accepts paths to local and remote files, which it then loads as an update to the application. It appears that the developer expects the files to be provided by the browser as part of an update

mechanism from their website; in reality, any application can supply the path to the file. The third crashes when any of its public Activities are loaded by other applications. The fourth has a public Activity that does not appear to be a useful addition to any other application.

8.3 Ease of Web Development

We discuss IPC Inspection from the perspective of a web developer and give examples of how it would be applied.

Deputies. Web applications that want to accept `postMessages` without risking privilege reduction should register a list of trusted requesters. It is already best practice to check the origin of message senders [29]; we make this logic explicit by providing a mechanism to register a list of acceptable requesters with the browser. Even if a message does cause a permission failure, web applications should already be built with the expectation that access to a device API might fail because users already expect to continue interacting with websites after denying permissions.

Any web application, regardless of whether it intends to act as a deputy, may be multiply instantiated because any website can be opened by another web site. Web applications already expect to be simultaneously open in multiple tabs in the same browser.

IPC Inspection does impose one restriction on web applications. If `a.com` opens the child frame `b.com`, and `b.com` in turn opens a child frame `a.com`, we place the two versions of `a.com` in separate instances because they have different requesters.[2] The two instances of `a.com` can obtain references to each other's `window` objects [25], which we support. However, the Same Origin Policy implies that they should have full access to each other's DOM objects, but IPC Inspectiondisallows this interaction because they are separate instances. We are aware of only one legitimate use of this embedding pattern, which is to facilitate cross-origin communication between two sites in browsers that lack `postMessage` support. However, modern browsers that support device APIs will also support `postMessage`, obviating the need for the embedding.

Requesters. Requesters do not need to make any changes to their applications because browser device permissions are currently all time-of-use. When a deputy makes an API call on behalf of a requester, the browser will display a time-of-use prompt that asks the user to grant the permission to the deputy and requesters.

[2]We do not break the case where `a.com` opens two child frames from `b.com`; both will be placed in the same instance and will have access to each other's heaps as expected.

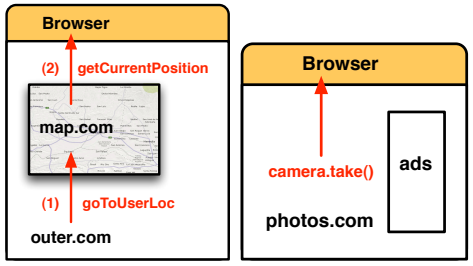

Figure 8: Left: an unprivileged website opens a mapping service that uses the user's location. Right: a privileged website includes unprivileged advertisements.

Examples. Both `postMessage` and device APIs are too new for widespread support and use, so we cannot measure the impact of IPC Inspection on real-world applications. Instead, we present two example cases of applications interacting with each other (Figure 8).

In the first example, a website (`outer.com`) opens a mapping service. `Outer.com` has no permissions, so the mapping service is opened as a new instance with no permissions. When `outer.com` asks `map.com` to display the user's current position on the map, `map.com` asks the browser for the current location. The browser would then prompt the user to give `outer.com` and `map.com` access to the current location.

In the second example, the user has granted camera access to a photo sharing website. The photo sharing website loads a frame containing advertisements. The ad site does not have any permissions, so it does not lose any permissions when it is opened. The photo sharing website can send messages to the advertising site without any changes to either party's permissions. However, a `postMessage` from the advertising site to `photos.com` would remove `photos.com`'s camera permission. If `photos.com` were to use the camera again after receiving a message from `ads.com`, the user would be presented with a prompt asking to approve camera access for both `photos.com` and `ads.com`. If the photo sharing site wishes to avoid becoming a deputy, it should refuse `postMessages` from the advertisement. If it needs to receive *replies* from the ad, it can use our proposed request-reply variant of `postMessage` when communicating with the ad (Section 6.2).

8.4 Performance

The performance cost of IPC Inspection depends on the workload, i.e., the set of running applications.

No Deputies. If the workload does not contain any permission re-delegation, then there is no cost. This occurs when applications don't communicate, messages are only being sent to applications with no permissions, or the requesters have as many permissions as the deputies.

Singletons. A singleton is an application that is never duplicated and has only one set of current permissions. Top-level windows in the browser are singletons, as are self-identified singleton applications in Android. If privilege reduction applies to a singleton, then the cost of IPC Inspection is (1) removing permissions from a list or hash map and (2) adding the removed permissions to a hash map that records the reason for removal. Neither is an expensive operation.

Instances. The primary cost of IPC Inspection occurs when privilege reduction requires the creation of a new instance. In a browser, new instances are created for child frames. The frame needs to be opened, regardless of whether it is a new instance; the difference with IPC Inspection is that the browser gives the child frame a unique entry in the permission assignment map, separate from the main application. This is a small cost.

In Android, the creation of a new instance might mean that multiple versions of the same application are running simultaneously, in different processes and virtual machines. Given the battery and memory constraints of a mobile phone, this does not scale well. However, we do not expect many instances to be open simultaneously. The standard pattern for legitimate communication is as follows: (1) the requester opens a target Activity, (2) the user performs an action such as selecting a contact or approving an e-mail, (3) the target Activity closes and the requester regains control of the screen. The instance only needs to exist while the Activity is open. Only one Activity can be open at once, so we expect that in most cases only one additional instance would be open at a time.

9 Related Work

Browser Defenses. Major browser vendors remove the geolocation permission from iframes, so that the user must re-approve the geolocation permission for every parent-child window pair. This agrees with our proposal. However, we suggest that these rules also be extended to top-level windows that interact with each other.

Android. Three pieces of concurrent work address similar issues. Davi et al. discuss permission re-delegation attacks on Android [8]. They introduce the problem and present an attack on a vulnerable deputy. We perform a larger analysis of applications and discuss how platforms need to change to prevent these attacks. Chin et al. present ComDroid [6], a static analysis tool that aims to help prevent developers from accidentally making components public. They also make recommendations for changes to the Android platform to reduce the rate of unintentional deputies. Although their tool and their platform recommendations would help prevent some instances of permission re-delegation, attacks on intentional deputies would still remain. Dietz et al. built Quire [10], an extension to the Android IPC mechanisms that helps developers avoid permission re-delegation attacks. Quire annotates IPCs so that an application can check the full chain of applications responsible for an IPC call. This addresses the same problem as IPC Inspection but does not force developer compliance.

Past work has also discussed Android permission usage. TaintDroid [12] performs dynamic taint analysis. It tracks the real-time flow of sensitive data through applications to detect inappropriate sharing. The taint source is API data, and the network is the sink. They track only data flow, but not control flow. TaintDroid is complementary to IPC Inspection because they track API return values but do not prevent API calls from being made. Another tool, ScanDroid [21], uses static analysis to determine data flow through Android applications; it is intended for use similar to TaintDroid. ScanDroid, however, requires access to application source code. Kirin [13] checks application permission requirements and recommends against the installation of applications with certain permission combinations. Their rules are intended to help detect malware, and they do not consider application interaction as a capability.

HBAC. IPC Inspection revises and extends History-Based Access Control (HBAC) [1]. The two approaches share a core idea: application permissions are reduced after inter-application interactions. HBAC is intended for use within a runtime; their permissions apply to threads, and privilege reduction follows function calls. We apply IPC Inspection to the application platform itself and create rules appropriate for that context. Our design places a high priority on application functionality and ease of development. Unlike HBAC, we do not reduce privileges following return values or permit explicit rights restoration. We introduce the concept of multiple instances of an application to prevent privilege reduction from impacting the application as a whole.

Stack Inspection. IPC Inspection has the same semantics as stack inspection, but permission checks are associated with IPC rather than method calls. We do not depend on the stack, so event-driven code does not present a problem. We make message queues explicit and external, by servicing each message with a new application instance. IPC Inspection is also runtime- and language-independent.

DIFC. Decentralized information flow control (DIFC) lets applications explicitly express their information flow policies to the operating system or a language runtime, which then enforces the policies [30, 26, 39, 11]. DIFC is not suitable for the problem of permission re-delegation because the access control policy for user-controlled resources is centrally decided by the user, not applications.

IPC Inspection is more similar to centralized information flow control, but IPC Inspection is deployed at the application level rather than the variable level.

Low Watermark. IPC inspection carries out the semantics of Biba's low watermark model [4] in that subjects are application instances, and the integrity level of an application instance is determined by the permissions that the user has granted to the application instance. When Instance A sends an IPC message to Instance B, the message represents objects with the same integrity level as that of Instance A. If Integrity(A) $<$ Integrity (B) (meaning B contains permissions that A does not), then B removes the permissions that A does not have.

LOMAC [20] applies Biba's low watermark mode in a different way. LOMAC aims to prevent (malicious) low integrity content from tampering with high integrity program execution, whereas IPC Inspection is intended to prevent less-privileged (low integrity) applications from using the additional privileges belonging to another (high integrity) application. In LOMAC, a subject is a job (which contains multiple application instances) and an object is data. Integrity levels for objects are assigned based on the sources for the objects. For example, Internet objects are at a lower integrity level than local data. Named pipes and shared memory are considered objects with integrity levels. Subjects' integrity levels are assigned based on the hierarchy of the jobs. The first set of system jobs have the highest integrity level and lower levels in the job hierarchy (as jobs spawn new jobs) represent lower integrity levels. Both LOMAC and IPC inspection face the "self revocation problem" [20] inherent in the Biba's low watermarking model. The self revocation problem occurs when a principal's privilege reduction prevents it from accessing high-integrity level data, preventing legitimate functionality. Each scheme has to relax the low watermark model slightly to accommodate the problem. In our case, we ignore the reply in the non-simplex IPC communications. In the case of LOMAC, they use jobs as subjects rather than processes.

CSRF. Like permission re-delegation, cross-site request forgery (CSRF) is a confused deputy attack that occurs in browsers [40]. However, CSRF attacks are targeted at server-side resources. CSRF defenses rely on developer participation and require changes to servers [2].

10 Conclusion

We discuss permission re-delegation as a problem with new permission systems. Permission re-delegation occurs when a deputy delegates a user-controlled permission to an unprivileged application without user authorization. This is an emerging threat for both the web and smartphone platforms. We find that many Android applications are at risk of having permission re-delegation vulnerabilities, and we construct attacks that exploit 15 vulnerabilities in Android system applications. We disclosed our findings and filed bug reports; several of the vulnerabilities have been confirmed as bugs.

We also devise a runtime-independent defense mechanism, IPC Inspection, which transparently protects against attacks on confused deputies, with no compatibility cost for non-deputies or confused deputies. However, intentional deputies and their clients need some modifications to work with IPC Inspection. In particular, applications that interact with deputies may need to add permissions that they otherwise do not use. We feel that the problem of permission re-delegation deserves careful attention, and we hope this paper will encourage future work on these problems. In particular, we believe static analysis of deputies is a promising future area for server-side analysis or platforms with installed packages.

Acknowledgements

We would like to thank Dean Tribble, William Enck, and David Wagner for their insightful comments. This work is partially supported by National Science Foundation grant CCF-0424422. This material is also based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] ABADI, M., AND FOURNET, C. Access Control Based on Execution History. In *NDSS* (2003).

[2] BARTH, A., JACKSON, C., AND MITCHELL, J. Robust Defenses for Cross-Site Request Forgery. In *CCS* (2008).

[3] BARTH, A., WEINBERGER, J., AND SONG, D. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *USENIX Security* (2009).

[4] BIBA, K. J. Integrity Considerations for Secure Computer Systems. Tech. Rep. ESD-TR-76-372, USAF Electronic Sstems Division, Hanscom Air Force Base, 1977.

[5] CHEN, S., ROSS, D., AND WANG, Y.-M. An Analysis of Browser Domain-Isolation Bugs and A Light-Weight Transparent Defense Mechanism. In *CCS* (2007).

[6] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing Inter-Application Communication in Android. In *MobiSys* (2011).

[7] CLULEY, G. Windows Mobile Terdial Trojan makes expensive phone calls. `http://www.sophos.com/blogs/gc/g/2010/04/10/windows-mobile-terdial-trojan-expensive-phone-calls/`.

[8] DAVI, L., DMITRIENKO, A., SADEGHI, A., AND WINANDY, M. Privilege escalation attacks on Android. In *ISC* (2010).

[9] DEPARTMENT OF DEFENSE. Trusted Computer System Evaluation Criteria (Orange Book). DoD 5200.28-STD, December 1985.

[10] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *USENIX Security* (2011).

[11] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLAR, D., KOHLER, E., MAZIERES, D., KAASHOEK, F., AND MORRIS, R. Labels and Event Processes in the Asbestos Operating System. In *SOSP* (2005).

[12] ENCK, W., GILBERT, P., CHUN, P., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI* (2010).

[13] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On Lightweight Mobile Phone Application Certification. In *CCS* (2009).

[14] ERMOLINSKIY, A., KATTI, S., SHENKER, S., FOWLER, L., AND MCCAULEY, M. Towards Practical Taint Tracking. Tech. Rep. UCB/EECS-2010-92, UC Berkeley, 2010.

[15] FELT, A. P. DeskClock service should require WAKE_LOCK permission. http://code.google.com/p/android/issues/detail?id=14659.

[16] FELT, A. P. MediaPlaybackService should require WAKE_LOCK permission. http://code.google.com/p/android/issues/detail?id=14660.

[17] FELT, A. P. PhoneApp Receiver can be abused. http://code.google.com/p/android/issues/detail?id=14600.

[18] FELT, A. P. Settings app – security bug. http://code.google.com/p/android/issues/?id=14602.

[19] FOURNET, C., AND GORDON, A. D. Stack inspection: theory and variants. In *POPL* (2002).

[20] FRASER, T. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *IEEE Symposium on Security and Privacy* (2000).

[21] FUCHS, A., CHAUDHURI, A., AND FOSTER, J. S. SCanDroid: Automated Security Certification of Android Applications. Tech. rep., University of Maryland, College Park, 2009.

[22] GOOGLE INC. Android 2.2 Compatability Definition. http://static.googleusercontent.com/external_content/untrusted_dlcp/source.android.com/en/us/compatibility/android-2.2-cdd.pdf.

[23] HARDY, N. The Confused Deputy: (or why capabilities might have been invented). In *ACM SIGPOS Operating Systems Review* (1988), vol. 22.

[24] HICKSON, I. HTML Device: An addition to HTML. http://dev.w3.org/html5/html-device, 2010.

[25] HICKSON, I. HTML5: Loading Web Pages: Browsing Contexts. http://dev.w3.org/html5/spec/browsers.html#windows, November 2010.

[26] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, F., KOHLER, E., AND MORRIS, R. Information Flow Control for Standard OS Abstractions. In *SOSP* (2007).

[27] LEOPANDO, J. TrendLabs Malware Blog: New Symbian Malware on the Scene. http://blog.trendmicro.com/new-symbian-malware-on-the-scene, June 2010.

[28] MOZILLA. Content Security Policies (CSP). https://wiki.mozilla.org/Security/CSP/Specification.

[29] MOZILLA. window.postMessage. https://developer.mozilla.org/en/DOM/window.postMessage.

[30] MYERS, A., AND LISKOV, B. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology 9*, 4 (2000), 410–442.

[31] PALLER, G. Dedexer. http://dedexer.sourceforge.net/.

[32] POPESCU, A. Geolocation API Specification. http://dev.w3.org/geo/api/spec-source.html.

[33] RUDERMAN, J. The Same Origin Policy. http://www.mozilla.org/projects/security/components/same-origin.html.

[34] SERIOT, N. iPhone Privacy. *Black Hat DC* (2010).

[35] W3C. Device APIs and Policy Working Group. http://www.w3.org/2009/dap/.

[36] WALLACH, D. S., AND FELTEN, E. W. Understanding Java Stack Inspection. In *IEEE Symposium on Security and Privacy* (1998).

[37] WANG, H. J., FAN, X., HOWELL, J., AND JACKSON, C. Protection and Communication Abstractions in MashupOS. In *ACM Symposium on Operating System Principles* (October 2007).

[38] WANG, H. J., MOSHCHUK, A., AND BUSH, A. Convergence of Desktop and Web Applications on a Multi-Service OS. In *Usenix Workshop on Hot Topics in Security* (2009).

[39] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIERES, D. Making information flow explicit in HiStar. In *OSDI* (2006).

[40] ZELLER, W. P., AND FELTEN, E. W. Cross-Site Request Forgeries: Exploitation and Prevention. Tech. rep., Princeton University, 2008.

[41] ZXING. Projects and Products Using ZXing. http://code.google.com/p/zxing/wiki/InterestingLinks.

Appendix

The 16 pre-installed system applications referenced in Section 4 are: Browser, Calendar, Calculator, Camera, Contacts, Desk Clock, Email, Gallery, Global Search, Launcher, Live Wallpaper, Messaging/Mms, Music, Phone, Settings, and SoundRecorder. These pre-installed applications must be present on every phone, as set forth by the Android 2.2 compatibility definition [22]. We built permission re-delegation attacks using Settings, DeskClock, Phone, Music, and Launcher. We collected the applications from the Android Market on August 27, 2010 (free) and October 15, 2010 (paid).

The 20 applications surveyed in Section 8.2 are: Daum Maps, Pages Jaunes, Korean IME, Sherpa, Qik, Yandex Maps, First Aid, Three Stooges, Öffi, Cheech and Chong, Coupons, Human Body Facts, Android System Info, Baidu Input, Bubbles, Hello Kitty Wallpaper, Musical Lite, Time2Hunt Free, ModernInfo: BlackOps, and Wolfram Alpha.