Towards Comprehensible and Effective Permission Systems

Adrienne Porter Felt

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-185
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-185.html

August 9, 2012

Towards Comprehensible and Effective Permission Systems

by

Adrienne Porter Felt

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor David Wagner, Chair
Professor Vern Paxson
Professor Tapan Parikh

Fall 2012

Towards Comprehensible and Effective Permission Systems

by

Adrienne Porter Felt

Abstract

Towards Comprehensible and Effective Permission Systems

by

Adrienne Porter Felt

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David Wagner, Chair

How can we, as platform designers, protect computer users from the threats associated with malicious, privacy-invasive, and vulnerable applications? Modern platforms have turned away from the traditional user-based permission model and begun adopting application permission systems in an attempt to shield users from these threats. This dissertation evaluates modern permission systems with the goal of improving the security of future platforms.

In platforms with application permission systems, applications are unprivileged by default and must request permissions in order to access sensitive API calls. Developers specify the permissions that their applications need, and users approve the granting of permissions. Permissions are intended to provide *defense in depth* by restricting the scope of vulnerabilities and *user consent* by allowing users to control whether third parties have access to their resources.

In this dissertation we investigate whether permission systems are effective at providing defense in depth and user consent. First, we perform two studies to evaluate whether permissions provide defense in depth: we analyze applications to determine whether developers request minimal sets of permissions, and we quantify the impact of permissions on real-world vulnerabilities. Next, we evaluate whether permissions obtain the user's informed consent by surveying and interviewing users. We use the Android application and Google Chrome extension platforms for our studies; at present, they are popular platforms with extensive permission systems.

Our goal is to inform the design of future platforms with our findings. We argue that permissions are a valuable addition to a platform, and our study results support continued work on permission systems. However, current permission warnings fail to inform the majority of users about the risks of applications. We propose a set of guidelines to aid in the design of more user-friendly permissions, based on our user research and relevant literature.

Professor David Wagner
Dissertation Committee Chair

Contents

Acknowledgements

This thesis would not have been possible without my family, friends, and colleagues. I would like to express my thanks to everyone who has provided me with support.

I owe my deepest gratitude to my family. My family has always encouraged me to be curious, tenacious, and ambitious. I attribute my career in computing to my upbringing; at present, women are underrepresented in the field of Computer Science, but the lessons I learned at home taught me not to be intimidated by the gender imbalance. My father brought me to his software company's office, my grandfather patiently answered all of my grade school "interview" questions about his career, and my mother never accepted my excuses for not completing my math homework. I was always told that I could accomplish anything.

I am grateful to my husband Mark, who always provided me with his patience and support. Without him, I surely would have starved while working on paper deadlines.

I would like to thank my graduate advisor, Professor David Wagner. His keen advice taught me how to formulate problems and identify feasible research directions. Too often, academia is harsh and hypercritical; David taught me how to temper criticism with kindness and a genuine desire to help. I also would like to thank my undergraduate advisor, Professor David Evans, who introduced me to research and encouraged me to consider graduate school. I would not have applied to graduate school without his assurance that I was capable of completing a Ph.D.

Professors Vern Paxson, Coye Cheshire, and Tapan Parikh provided valuable feedback on my work that helped shape my research. Their comments contributed greatly to the way I thought about the problems discussed within this dissertation.

I am also indebted to Adam Barth, who served as my unofficial mentor during my first year of graduate school. He taught me how to write a research paper.

Chapter 1

Introduction

Most modern platforms support large, thriving third-party application marketplaces. Users can select from an unprecedented number of applications to supplement their experiences with their smartphones, computers, web browsers, and social networking sites. For example, as of 2012, Google Play lists over 500,000 Android applications, the Facebook platform supports more than nine million applications, and the Apple App Store includes more than 600,000 iOS applications [106, 53]. These applications offer a diverse set of features from a diverse set of developers.

Unfortunately, third-party applications create risks for the user. Many otherwise legitimate applications aggressively collect personal information about their users (e.g., for marketing campaigns) in ways that make users uncomfortable [144]. Malicious applications use social engineering tactics to convince users to install them [64, 159, 48, 134]. Applications can also put users at risk of external (e.g., network-based) attacks by containing vulnerabilities; the authors of third-party applications usually are not security experts [104, 154]. How can platform designers help users avoid these threats while still supporting a broad range of colorful applications?

Traditional user-based security mechanisms were designed to protect users from each other on time-shared computers, in an era when most applications were downloaded from trusted sources or written by users themselves. Consequently, traditional operating systems assign the user's full privileges to all applications. However, this threat model is no longer appropriate now that users cannot fully trust their applications. Modern platforms are consequently transitioning to a new security model in which each application has a different set of permissions based on its requirements. These permissions control applications' access to security- and privacy-relevant system resources, so that users can decide whether to give individual applications access to these sensitive resources.

This new style of permission system can be found in contemporary smartphone operating systems (e.g., Android and Windows Phone 7), new desktop operating systems (e.g., Windows 8 Metro), social networking platforms (e.g., Facebook and Twitter), and browsers (e.g., the Google Chrome extension platform and new HTML5 features). In some of these platforms, users are prompted to approve permissions as needed by applications at runtime. In others, developers are asked to declare their applications' permission requirements up-front so that users can grant per-

missions at install-time. Regardless of when the permission request occurs, users are asked to make security decisions on a per-application basis, and developers need to design their applications to work within the constraints of the permission systems.

This dissertation evaluates whether modern permission systems have a positive effect on end user security, with the goal of guiding the design of security mechanisms in future platforms. Application permissions can potentially provide two security benefits:

- **Defense in Depth:** For systems with install-time permissions, the impact of a vulnerability in an application will be limited to the vulnerable application's declared permissions. This could also be true for a system with runtime consent dialogs, if developers declare their applications' maximum possible permissions up-front.
- **User Consent:** Security-conscious users may be hesitant to grant access to dangerous permissions without justification and trust in the application. Ideally, permissions should empower users to provide or withhold their informed consent.

We use the Android application and Google Chrome extension permission systems as case studies to evaluate whether modern permission systems are achieving these goals. We selected these platforms for case studies because they are popular platforms with extensive permission systems, relatively thorough documentation, and available source code.

Specifically, this dissertation investigates the following questions:

- *Do developers request fewer than full privileges?* A low permission request rate is a prerequisite for achieving the defense in depth and user consent benefits. Permissions can reduce the severity of vulnerabilities only in applications without high-severity permissions, and users would be overwhelmed by large numbers of consent dialogs or warnings. We evaluate the Google Chrome extension and Android application permission request rates in Chapter 4.
- *Do permissions mitigate vulnerabilities?* An overall low permission request rate is not sufficient to achieve defense in depth. More specifically, vulnerable applications need to have low permission request rates. In Chapter 5, we identify real-world vulnerabilities in extensions and consider whether the extensions' permissions mitigate them.
- *Do permissions inform users about the risks of applications?* Users need to pay attention to, comprehend, and act on permission information. We performed two user studies — a large-scale online survey and a set of in-person interviews — to determine whether Android's install-time permission warnings are usable security indicators that inform users (Chapter 6). We also investigate whether the permissions that platforms have chosen to ask users about match the resources that users are concerned about protecting from applications (Chapter 7).

We conclude that permissions are a valuable addition to a platform because they reduce the impact of vulnerabilities in applications if developers declare them up-front. Based on this finding, we strongly recommend the inclusion of permissions with up-front declarations in future platforms. However, Android's install-time permission warnings fail to adequately inform the majority of users about the risks of applications. Future permission systems need to investigate different ways of presenting permissions to users.

With the goal of usable permissions in mind, Chapter 8 proposes a set of guidelines to aid future platform designers. When should the platform ask the user for consent? What types of permissions need consent? How should consent dialogs be presented to the user? We aim to answer these questions by discussing the strengths and weaknesses of the different design choices that are available to platform designers. Existing permission systems each rely primarily on a single type of permission-granting mechanism for obtaining consent from users. Instead, we recommend that platforms should include multiple permission-granting mechanisms, assigned to and customized for permissions based on the nature of each specific permission. We apply our proposal to a set of platform-neutral operating system permissions, and a preliminary evaluation indicates that our proposed model has the potential to improve the user experience.

We believe that the results of our case studies are generalizable to other platforms. Our evaluations of the Android and Google Chrome platforms were structured to generate both platform-specific and platform-neutral lessons and recommendations. Furthermore, burgeoning platforms (e.g., Boot2Gecko, the HTML5 Device API, the Mozilla WebAPI) are being directly influenced by Android and Google Chrome's permission systems. Consequently, we believe that our recommendations and proposals apply to future platforms as well as Android and Google Chrome.

Chapter 2

An Overview of Approaches to Permissions

We discuss the evolution of permission systems and warnings, from the early time-sharing systems to modern smartphones and browsers. We also survey the different types of platforms with user-facing permissions. This serves to contextualize this dissertation.

2.1 Permissions For Malicious Users

Through the 1980s, most computers were time-shared by multiple users. Users sometimes spied on or harmed each other, which introduced the security threat of a malicious user. Users wrote many of the programs that ran on their computers, and the first widespread malware did not appear until 1988 [59]. Consequently, operating system security mechanisms were designed to protect users from each other, and applications were assumed to run on the behalf of their invoking users. Their primary security goal was to isolate users, with controlled sharing between users.

UNIX, VAX/VMS, and Multics are three prominent examples of time-sharing operating systems. They were built with the threat of a malicious user in mind. UNIX associates protection bits with each file that specify which users may read, write, and execute a given file; when a program executes, it typically can only access the files that the invoking user can access [124]. In VAX/VMS, users are assigned to one of seven privilege levels, and processes run with their invoking users' privilege levels unless otherwise specified by a system administrator [92]. Multics processes similarly run on behalf of human users [128]. Indeed, an external evaluation of Multics by U.S. Air Force officers found that Multics had the potential to enforce user isolation well enough to store the military's classified files [86, 87]. Modern desktop operating systems (e.g., Windows, Mac OS X) inherited UNIX's access control model; as a result, these platforms still typically grant all of a user's permissions to the user's applications.

Within this context of malicious users, Saltzer and Schroeder famously defined seven design principles: economy of mechanism, fail-safe defaults, complete mediation, open design, privilege separation, least privilege, least common mechanism, and psychological acceptability [127]. These principles provided the foundation of modern computer security. This dissertation focuses on whether permission systems facilitate least privilege and psychological acceptability, which Saltzer and Schroeder define thusly:

> Least privilege: Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error.
>
> Psychological acceptability: It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.

Although Saltzer and Schroeder originally defined these principles for security mechanisms that protect users from each other, they are still relevant in the modern context of malicious applications.

2.2 Permissions For Adversarial Applications

As the popularity of computing increased, malicious and benign-but-buggy applications became a concern. Users could be tricked into installing malicious applications, applications could take advantage of operating system vulnerabilities to remotely install themselves, or poorly written applications could enable external attackers to take control of a computer. When this shift occurred, applications could no longer be thought of as trusted agents for users. It became inappropriate for applications to have all of the privileges of their invoking users. In response, researchers developed mechanisms for restricting the privileges of applications. However, the focus was on the security mechanisms themselves; little effort was made to make the policies developer- or user-friendly.

Omniware. Omniware is a mobile code system that provides safety by isolating and controlling untrusted mobile code [21]. It uses software fault isolation [150] to isolate applications from each other despite sharing an address space. The authors of Omniware focused on isolation techniques and left the details of policies and policy generation to future work.

System Call Interposition. Several subsequent monitoring systems enforced policies that restricted applications' system calls. These policies are inherently complex and difficult to write because of their detailed, low-level, technical nature. Among the first of these systems was TRON: a user could use TRON to execute a process in a restricted domain with a subset of the user's file permissions [33]. In order to use TRON, the user would need to invoke the process with a list of all of the files and directories that the process needs access to. Janus, Systrace, and Ostia allow a user to whitelist the system calls that individual applications could make [73, 121, 70]. Janus and Ostia policies are at the level of individual system calls and files. The authors of Systrace

attempted to simplify the policy creation process with a policy generation tool that creates policies from execution traces. Dynamic analysis cannot automatically exercise all of an application's paths, so Systrace's policy generation tool is supplemented with an interactive policy mode for when Systrace encounters a system call that the trace-generated policy does not cover. Although the policy generation tool was an improvement, it was still complex and unfriendly to users who do not know what system calls are.[1]

Mandatory Access Control. Mandatory access control (MAC) systems 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. MAC systems have been deployed to restrict the damage of both malicious users (e.g., military users with different security clearances [14]) and adversarial applications (e.g., Internet-connected applications with bugs [67]).

SE Linux is a version of Linux with mandatory access control to limit the damage of benign-but-buggy applications [10]. Under SE Linux, applications are supposed to be restricted to least privilege. System administrators and distribution managers are supposed to write policies to identify the minimal sets of permissions that applications require. However, the policies are difficult to author, even for system administrators and power users. For example [49], consider the following excerpt from a policy:

```
full_user_role(staff)
allow staff_t unpriv_userdomain:process signal_perms;
can_ps(staff_t, unpriv_userdomain)
```

This short example demonstrates the complexity of defining policies: policy administrators need to manage the creation and assignment of roles, domains, and processes. Due to this complexity, many applications do not have appropriate policies. Incorrect or missing policies will break applications. Consequently, SE Linux is typically run in "targeted mode," in which only certain high-risk applications are subjected to SE Linux's restrictions.

POSIX Capabilities. POSIX capabilities aim to reduce the need to run applications as the root user [78]. Instead of having a single, all-powerful root user, the system supports capabilities that each endow the ability to perform a small number of privileged tasks. Using POSIX capabilities, applications are invoked with the capabilities that they need; although each individual capability is potentially dangerous, a malicious or vulnerable application still has fewer privileges with POSIX capabilities than as a traditional root user. Using POSIX requires that the user or developer assign the correct capabilities to the application, which requires in-depth knowledge of the operating system. POSIX also suffered from a series of design and implementation security flaws [115].

Capsicum. Capsicum uses capabilities to protect users from vulnerabilities in benign-but-buggy applications [151]. Capsicum capabilities are rights to objects (i.e., files or sockets). Unlike UNIX, Omniware, POSIX capabilities, and many other file access control systems, Capsicum's privileges

[1] For reference, Provos included a screenshot of the tool in the paper that describes Systrace [121].

are more fine-grained than read or write (e.g., `seek` or `fstat`). Developers only have to make minor changes to their applications to correctly obtain their capabilities. However, users are not involved in the capability approval process: obtaining user consent was not one of their goals. Like other work in this vein, their policies are too low-level to easily ask end users about.

Model-Carrying Code. With all of the previously described systems, it is unknown whether an application will attempt restricted actions until runtime. In contrast, model-carrying code (MCC) verifies an application's safety properties prior to installation [132]. With MCC, a model extraction tool creates a model of an application that describes the security implications of the given application. Adherence to the model is enforced at runtime. Users can compare the model against their own security policies to determine whether they wish to run the application. Policies are described with finite state automata that can express policies like "program P is allowed to use the function `gethostbyname`." This approach has the benefit of being developer-friendly (model extraction is automated), but users still need to create technical policies in a technical policy language.

2.3 Usable Permissions

Once isolation techniques had been established, researchers and companies began to consider how to make their policies usable by end consumers and developers.

2.3.1 Platforms With User-Facing Permissions

We overview several of the most notable platforms with user-facing permission systems. We later discuss user studies of those platforms in Chapter 2.3.2.

Java. Java applets were among the first instances of mobile code. They were commonly distributed as part of web sites. By default, applets are unprivileged. There have been several ways for applets to gain access to additional privileges [137]; the exact mechanism depends on the browser and version of Java. In some cases, users have been involved in the permission-granting process. Figure 2.1(a) shows a Java applet permission request in Netscape from 1999.

CapDesk. CapDesk is a capability-secure desktop [140, 101]. By default, applications have no privileges. In CapDesk, some permissions (capabilities) require user approval or interaction. CapDesk applications typically request permissions when they are launched, but applications can request additional permissions at runtime. Developers need to request capabilities from a central "powerbox" that distributes the capabilities to the parts of the application that require them.

Symbian. Symbian is a smartphone operating system; it was among the first smartphone operating systems to support the installation of arbitrary third-party applications. Symbian offers an application-signing service, and only signed applications are allowed to access the most dangerous Symbian permissions [83]. Signed applications automatically receive all of their requested permissions without user approval. However, Symbian prompts users to grant permissions during

the installation of unsigned applications (Figure 2.1(b)). Unsigned applications can only request a few permissions, like connecting to the network; the remainder are off-limits for unsigned applications. Developers need to determine the permissions that their applications require and, if necessary, apply for the Symbian signing process.

Blackberry. Blackberry is a smartphone operating system that supports third-party applications. When a user installs a Blackberry application, the user is asked whether to grant the application "trusted app status." A trusted application can request and receive permissions without prompting the user, although the user can modify the granted permissions in Blackberry's settings. Non-trusted applications have to prompt the user to gain specific permissions (Figure 2.1(c)). Blackberry applications can ask for permissions at any time, although most ask when the application is first launched. Blackberry permissions cover phone settings, user data, Bluetooth, WiFi, etc. [126]. Developers need to initiate permission requests before trying to use restricted API calls.

Windows UAC. In Windows Vista and Windows 7, users are encouraged to run applications from a low-privilege user account [111]. Applications run with the user's full privileges, but the low-privilege user account does not have many privileges. User account control (UAC) allows applications to ask for additional permissions as needed. With UAC, a permission dialog appears whenever an application attempts to perform a privileged action (Figure 2.1(d)). In order to use UAC, a developer must list the UAC resources that her application needs in a manifest file; a permission dialog will later be raised when the application tries to access the resource.

Facebook. Facebook supports a third-party application market. With users' approval, applications can access users' profile data, friends, and "activity feeds" [9]. Applications can ask for permission at any time, although most ask for permissions as part of installation. Figure 2.1(e) shows a permission request; the permissions are in the lower right quadrant of the dialog. Developers need to raise the appropriate permission request dialogs prior to trying to access protected API calls.

iOS. iOS is a smartphone and tablet operating system. iOS users can run third-party applications that they download from the Apple App Store. Apple subjects all iOS applications in the App Store to an official review process which includes a human security review. Due to this review, applications can access most privileges without OS-mediated user approval. However, some privileges require explicit user permission. In iOS versions 5 and earlier, user-controlled permissions governed access to GPS location and notifications (Figure 2.1(f)). In iOS 6, permissions are also required for contacts, calendar, and the photo library [74]. iOS permission prompts are raised automatically the first time that the developer tries to access the restricted resource.

Bitfrost. The One Laptop Per Child (OLPC) project distributes low-cost laptops to children in developing countries. Bitfrost is the security model for the OLPC operating system [96]. With Bitfrost, applications are unprivileged by default. Applications have to request privileges from the user. Permissions can be requested when the application is launched or at runtime.

Browsers. Web sites are sandboxed in the browser: by default, they cannot access local files or other user data. However, modern browsers are beginning to offer additional resources to web sites (e.g., the HTML 5 device API proposal [15]). Figure 2.2(a) shows the location dialog that appears when web sites attempt to read the user's current location.

Google Chrome Extensions. Google Chrome's extension platform allows third-party extensions to add functionality to the browser and modify web sites. In order to gain access to web sites or user data, Google Chrome extensions must request permissions. Developers list the permissions in a manifest file, and users approve the permission requests during installation (Figure 2.2(b)). The Google Chrome extension platform was the first browser extension platform to have a defined permission system; Firefox and Internet Explorer provide all extensions with full privileges. We selected Google Chrome as one of our case studies; it is described further in Chapter 3.

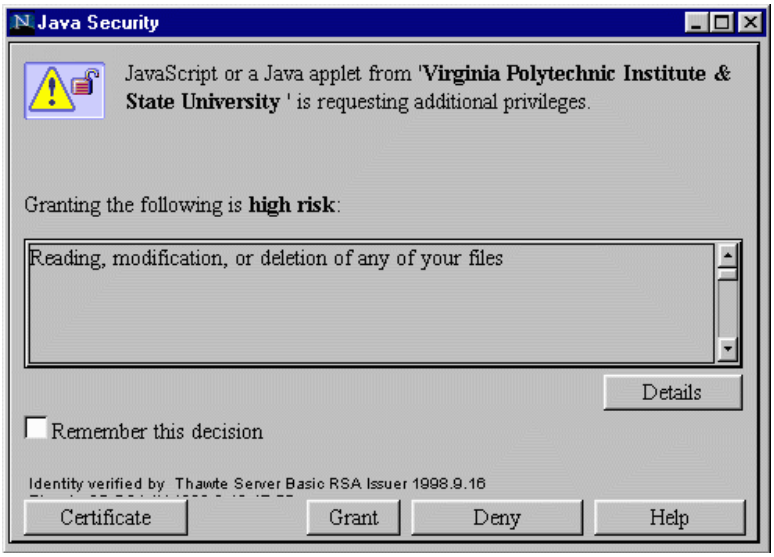
Android. Android is a smartphone and tablet operating system that supports third-party applications. By default, Android applications cannot access sensitive user data or phone settings. If an application needs such privileges, the developer must specify the list of permissions that the application requires. Android informs users of the application's desired permissions during installation (Figure 2.2(c)). Android's permission system is extensive. We selected Android as one of our case studies; its permission system is described further in Chapter 3.

Windows Phone. Beginning with Windows Phone 7, users are asked to approve install-time permissions for applications. Developers must specify the permissions that their applications need, and users are shown an application's permissions as part of the installation process. Applications cannot request additional permissions at runtime. Windows Phone 7 has 16 permissions, which cover hardware (e.g., the microphone) and personal data (e.g., phone identity). Windows 8 Metro-style applications are subjected to very similar restrictions: they are sandboxed by default, developers need to specify any additional permissions, and users are asked to grant permission at install-time. Figure 2.2(d) shows the permissions for the "Photos" application in Windows 8.

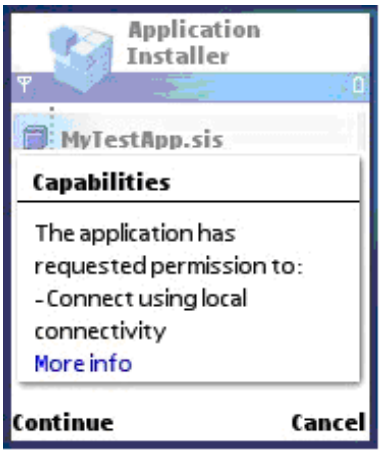
2.3.2 Warning Science

Users experience permissions as warnings about applications. We therefore can turn to warning science to understand how users interact with permissions.

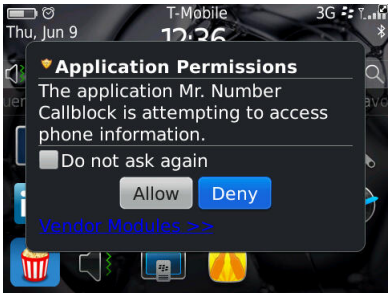
C-HIP. Wogalter proposed a model of how humans process warnings, known as the Communication-Human Information Processing (C-HIP) model [155]. The model formalizes the steps of a human's experience between being shown a warning and deciding whether or not to heed the warning. C-HIP assumes that the user is expected to immediately act upon the warning, which is appropriate for research on computer security dialogs. (Other warning science research has focused on situations in which consumers need to recall warnings for later use [107].) Researchers in the area of usable security have begun to use Wogalter's model to analyze the specific ways in which computer security dialogs can fail users. For example, Cranor used the C-HIP model as the basis for her "human in the loop" framework, which addresses problems for designers of interactive systems [51]. We similarly use the C-HIP model to study the usability of Android's permissions (Chapter 6).



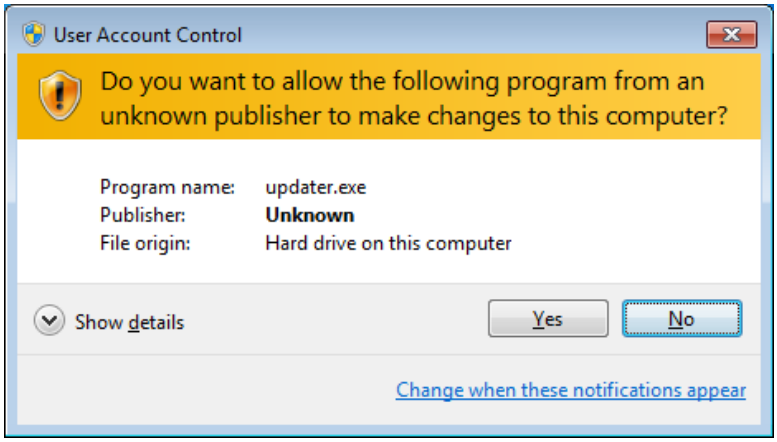
(a) Java applet, in a Netscape browser [136]



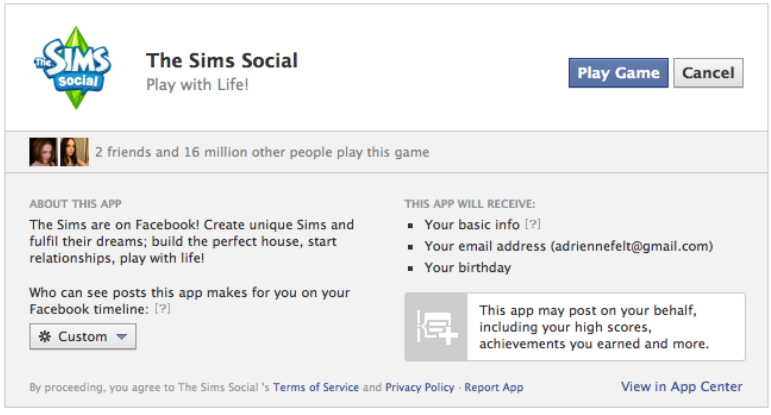
(b) Symbian [3]



(c) Blackberry [126]



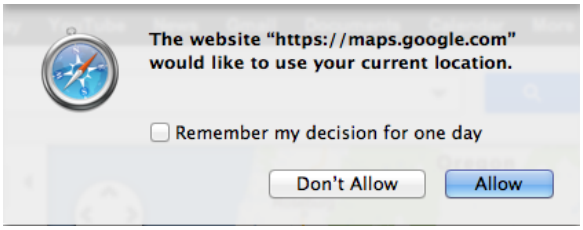
(d) Windows UAC



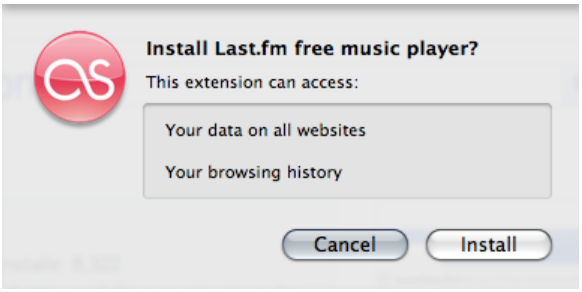
(e) Facebook

(f) iOS

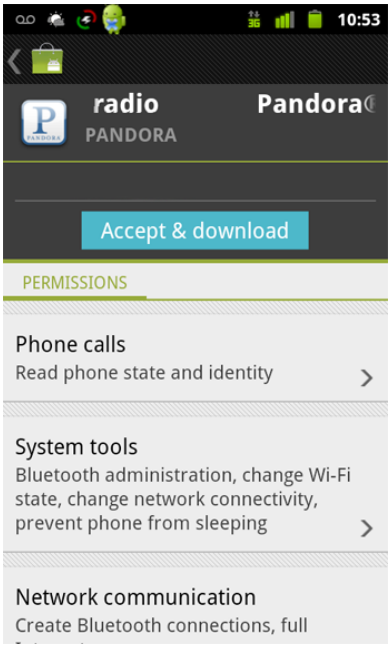
Figure 2.1. User-facing permission requests.

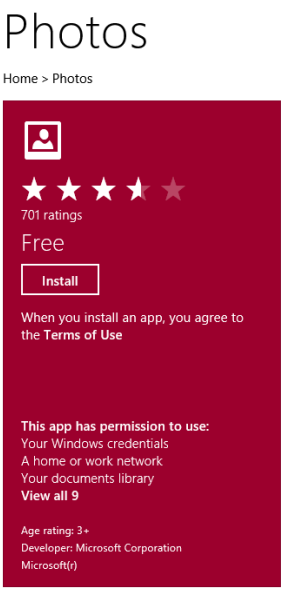


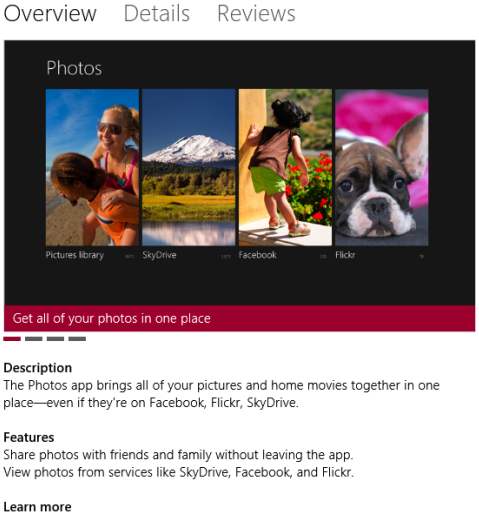
(a) Safari



(b) Google Chrome extensions







(c) Android

(d) Windows 8 Metro

Figure 2.2. User-facing permission requests.

Browser Security Warnings. Most warning research in the usable security community has centered on phishing and SSL certificate warnings. Although they serve a different purpose than permission warnings, the warnings are experienced similarly. Unfortunately, most research in this area indicates that users do not pay attention to or understand browser security warnings.

Dhamija et al. challenged desktop web browser users to identify phishing attacks in the presence of phishing and fraudulent certificate warnings [54]. 23% of their subjects completely ignored the passive phishing warnings, and 68% of subjects quickly clicked through the active fraudulent certificate dialogs. Egelman et al. used the C-HIP model to examine the anti-phishing warnings used by two popular web browsers [57]. They evaluated how people responded to phishing warnings when subjected to spear phishing attacks. Under their experimental conditions, 79% of subjects heeded the active (i.e., interruptive) warnings whereas only 13% heeded the passive (i.e., non-interruptive) warnings. Sunshine et al. performed a followup study using the C-HIP model to identify improvements to web browser certificate warnings [141]. Based on their subjects' low rates of attention and comprehension, their ultimate recommendation was to block access to dangerous sites without asking the user to try to understand certificate warnings. Together, these studies suggest that it is difficult to capture user attention for security. This leads us to believe that it will be similarly difficult to design usable permission warnings.

EULAs. Users often see End User License Agreements (EULAs) and terms of service (TOS) when installing applications. EULAs and TOS are similar to install-time permissions because they must be accepted as a condition of installation. Few people read EULAs or TOS. As an example, a software company offered a financial prize for anyone who read their EULA; only one person claimed it after four months and more than 3,000 downloads [108]. Good et al. asked people to select and install applications and found that most subjects ignored EULAs, few subjects knew what EULAs contained, and subjects often regretted their installation decisions after learning what was in the EULAs [75]. Unfortunately, users' lack of attention to EULAs carries over to other types of consent dialogs. Bohme et al. performed a large-scale field experiment that presented users with install-time consent dialogs, with varied degrees of resemblance to EULAs [41]. They found a positive correlation between the consent rate and how much the dialog resembled a EULA, even though all of the dialogs conveyed the same information. This indicates that permission system designers should design their warnings to not resemble EULAs or TOS.

Privacy Indicators. Researchers have examined whether the placement of privacy indicators on web sites or in search results will influence user behavior. Tsai et al. placed privacy icons in online shopping search results and found that subjects were willing to pay more to buy goods from online stores with good privacy practices [145]. In a follow-up study, Egelman et al. considered whether the timing and placement of privacy indicators affects user shopping choices [56]. They displayed privacy scores as part of search results, as a warning on the top of the page, or as a full-screen interruptive warning. Users who preferred to make a purchase from the first store they clicked on were influenced only by the icons in search results, whereas users who comparison shopped between stores were influenced by all three types of privacy indicators. Although we cannot draw direct comparisons between the online store search scenario and permissions, we can hypothesize that timing and placement will likewise be important for permission warnings.

Permission Warnings. Four other studies have considered the usability of permission warnings. Motiee et al. performed user experiments with Windows UAC dialogs [111]. When subjects encountered inappropriate (i.e., malicious) UAC prompts while completing other tasks, 68% of the subjects consented to the prompt or disabled UAC prompts completely. When subjects encountered inappropriate UAC prompts when installing an application, 95% of the subjects consented to the prompts. This suggests that users are not paying attention to Windows UAC dialogs, and installation may not be a good time for security decisions. However, newer permission warnings may be more effective: Android, iOS, and newer Windows permissions are simpler and more specific.

Besmer and Lipford conducted semi-structured interviews with Facebook users and found that users did not realize how much information they are sharing with Facebook applications [35]. King et al. ran a quantitative Facebook survey that asked subjects whether they noticed the Facebook permission dialog that appeared when entering the researchers' survey, and only a minority responded affirmatively [91]. However, this result is not necessarily generalizable; the participants knew the survey application had been created by a privacy researcher of their acquaintance, which may have altered their interest in security indicators. They also presented survey participants with general comprehension questions about the Facebook platform, such as whether Facebook applications are created by Facebook, and approximately half of participants answered each question correctly. (They do not report how many people answered all of the comprehension questions correctly.)

In concurrent work, Kelley et al. performed twenty semi-structured interviews to explore Android users' feelings about and understanding of permissions [90]. However, the scope of our usability studies (Chapters 6 and 7) is much broader: their study reported qualitative data and did not address attention, behavior, or users' concerns, whereas we collected large-scale quantitative results, performed an observational study, and experimentally measured comprehension with multiple metrics. Additionally, we designed our comprehension study to identity specific problems with the way permissions are presented.

Smartphone Privacy Concerns. Smartphone privacy research has traditionally focused on users' concerns about sharing location data, and numerous studies have examined the topic. Three independent studies found that the identify of the person with whom the user was sharing was the most important factor that influenced users' sharing decisions [50, 98, 153], whereas Barkhuus found that the user's current location matters more [28]. Barkhuus et al. also found that people like location services if they are useful and tend to stop worrying about location-based services after using them for a while [29, 28]. Anthony et al. observed 25 undergraduates for a week and found that they fell into three categories: people who never shared their location, people who always shared their location, and people who are willing to share depending on both the requester and the current location [25]. Kelley et al. report that 19 of 24 people "strongly agreed" that online companies should never share information like location unless it has been authorized by the user [89]. This prior literature thoroughly documents users' privacy concerns about sharing location data.

Smartphone operating systems provide applications with the ability to access a number of resources beyond location data, but few studies have explored the space of smartphone privacy and security beyond location. Roesner et al. studied smartphone users' privacy and security expectations for copy-and-paste, photography, and text messaging in addition to location; they found that most subjects thought that applications can only access these resources when the user initiates an action [125]. Muslukhov et al. asked a number of smartphone users about the value and sensitivity

of eleven types of data on their phones; their subjects report concern about sharing text messages, photos, voice recordings, contacts, passwords, e-mails, documents, and location [114]. Egelman et al. explored user attitudes towards Internet, location, audio, contacts, and photo permissions with two controlled economic experiments. Their experiments suggest that some users care about preserving the privacy of their contacts more than other types of data, but the experiments still only cover a small subset of permissions [58]. We aim to further expand the scope of research on users' smartphone concerns by studying users' opinions about 99 risks associated with 54 smartphone permissions (Chapter 7).

Smartphone Privacy Tools. Privacy researchers have built several tools to help Android users avoid privacy violations. Most research has focused on identifying malicious behavior [61, 68, 60, 62, 159, 116], without considering how to help users make informed security decisions. However, two sets of researchers have focused on usability. Howell and Schechter proposed the creation of a sensor-access widget, which visually notifies the user when a sensor like the camera is active [80]. Roesner et al. proposed user-driven access control: rather than asking users to review warnings, this approach builds permission-granting into user actions with trusted UI [125]. Neither of these past proposals are sufficient for all of the functionality currently in smartphone platforms; in Chapter 8, we build on this past work to discuss how to build a full smartphone permission system.

2.3.3 Developers and Permissions

End consumers are not the only users of permission systems: developers need to obtain the correct permissions for their applications. All but one of the permission systems described in Chapter 2.3.1 require the developer to raise a permission-granting dialog or list the required set of permissions.

Android Applications. Previous studies have examined Android permission usage, although they have been limited due to the lack of Android permission documentation. Enck et al. apply Fortify's Java static analysis tool to decompiled applications; they study their API use [60]. They looked for misuse of privacy sensitive information or developer errors that expose private information. The scope of their study could have been expanded if the documentation more clearly specified which API calls can be used to gain access to private information. Vidas et al. studied overprivilege in Android applications [147]. However, Vidas's results are not reliable because they solely relied on developer documentation when judging overprivilege; as we explain in Chapter 4.3.3, the documentation is too limited to use as a measure of correct permission usage.

Others have studied Android permissions without considering the role that developers play in permission requests. Enck et al. developed Kirin, a tool that reads application permission requirements during installation and checks them against a set of security rules [61]. They do not examine whether or how permissions are used by the applications. Barrera et al. examine 1,100 Android applications' permission requirements and use self-organizing maps to visualize which permissions are used in applications with similar characteristics [30]. They primarily focus on the structure of the permission system and conclude that some permissions are too broad.

Researchers at SMobile present a survey of the permissions requested by 48,694 Android applications [146]. They do not state whether their sample set is composed of free applications, paid applications, or a combination. They report that 68% of the applications in their sample set request enough permissions to be considered "suspicious." In Chapter 4, we similarly find that applications have high privilege requests and further explore why.

Google Chrome Extensions. Barth et al. conducted an analysis of 25 popular Google Chrome extensions that shows that most developers request fewer than all permissions [31]. However, their sample set is too limited and unrepresentative to be definitive: Google employees authored 9 of the 25 extensions, the most popular extensions might not be representative of all extensions, and the extension platform had only been public for a few weeks prior to their study. The results of our large-scale evaluation of Google Chrome extensions (Chapter 4.2) show that Barth et al.'s small-scale study overestimated the prevalence of extension privileges. Guha et al. [76] performed a concurrent, short study of the permissions used by Google Chrome extensions and report similar permission request rates to ours; however, they do not consider the effects of popularity. We provide a significantly more detailed discussion of extension privileges and consider the reasons for overprivilege.

Chapter 3

Case Studies' Backgrounds

We provide background on our two case studies: Android and the Google Chrome extension platform. We selected these platforms for our case studies because they are popular and widely-deployed platforms with established permission systems.

3.1 Android Applications

Android smartphone users can install third-party applications through the Android Market or Amazon Appstore. We present basic information about Android applications in Chapter 3.1.1. The quality and trustworthiness of these third-party applications vary widely, so Android treats all applications as potentially buggy or malicious. Each application runs in a process with a low-privilege user ID, and applications can access only their own files by default. In order to gain access to more privileges, applications must request permissions (Chapter 3.1.2). Users grant or deny these permissions during installation (Chapter 3.1.3). Android enforces permissions with proprietary mechanisms (Chapter 3.1.4).

3.1.1 Application Architecture

Android applications are split into components. Android defines four types of components: *Activities* to provide user interfaces, *Services* to run in the background, *Broadcast Receivers* to handle message passing, and *Content Providers* for data storage [1]. Applications can define many or none of each type of component. Applications are typically written in Java, but they can include native code. All of an application's code runs in the same process.

Android's *Intent* system is used for inter- and intra-application communication [6]. An Intent can be thought of as a self-contained object that specifies a remote procedure to invoke and includes the associated arguments. Applications commonly use Intents for internal communication between components [47], but applications also register for system-wide Intents. System-wide Intents are sent by the operating system to notify applications about events such as low battery or disconnection from the network. An application can make a component public by registering the component to receive Intents from other applications.

3.1.2 Requesting Permissions

Application developers determine the permissions that their applications require. An application may need permissions in order to access Android API calls, system-provided Content Providers, and system-wide Intents. Developers specify the appropriate permissions in files that are packaged with applications. Each application has its own set of permissions that reflects its functionality and requirements. There are 134 permissions in Android 2.2 for developers to select from. Android documentation categorizes permissions into threat levels:

- *Normal* permissions protect API calls with annoying but not harmful consequences. Examples include vibrating the phone and setting the wallpaper.
- *Dangerous* permissions protect potentially harmful API calls. These include actions that could cost the user money or leak private information. Example Dangerous permissions control opening a network socket, recording audio, and using the camera.
- *Signature/System* permissions protect the most sensitive operations. For example, these permissions protect the ability to reboot the phone or inject keystrokes into other applications. These permissions are difficult to obtain: Signature permissions are granted only to applications that are signed with the device manufacturer's certificate, and SignatureOrSystem permissions are granted to applications that are signed or installed in a special system folder. These restrictions essentially limit Signature/SignatureOrSystem permissions to preinstalled applications, and requests for Signature/SignatureOrSystem permissions by other applications will be ignored. However, advanced users who have rooted their phones [81] can manually install applications with SignatureOrSystem permissions.

As far as we are aware, permissions were classified into these categories without user input.

3.1.3 Install-Time User Consent

Users approve applications' permissions as part of the installation process. When a user initiates installation, a confirmation page interrupts the process to warn the user about the application's requested permissions (Figure 3.1). Dangerous permissions are displayed with three-layer warnings: a large heading that states each permission's general category, a small label that describes

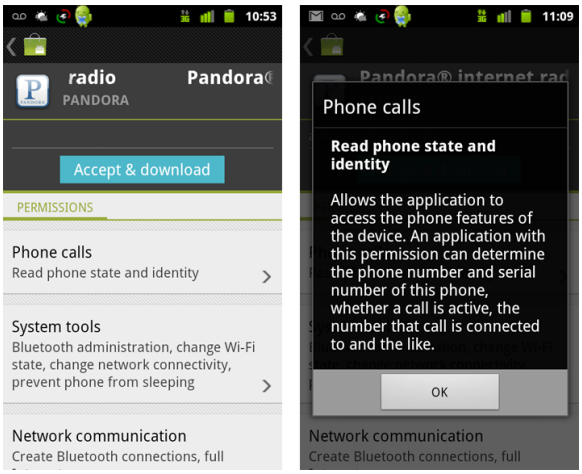


Figure 3.1. On the left, a screenshot of the Android Market's installation confirmation page, displaying the application's permission requests. On the right, the permission dialog that appears if a user clicks on a permission warning.

the specific permission, and a hidden details dialog that opens if the user clicks on the permission. If an application requests multiple permissions in the same category, their labels will be grouped together under that category heading. Normal and Signature/System permissions are hidden under a "More" label beneath the Dangerous permissions. At this point, users have a binary choice: they can accept all of the permissions and proceed with installation, or cancel the installation.

On most phones, Android users can download applications from both the official Android Market and non-Google stores such as the Amazon Appstore. When users install applications from the official Android Market, the confirmation page with permission information is displayed prior to payment and download. When users install applications from other sources, the confirmation page with permissions is always the last step of installation (i.e., after payment and download).

3.1.4 Permission Enforcement Mechanisms

The API. The Android API is composed of two parts: a library that resides in each application's virtual machine and an implementation of the API that runs in the system process(es). The API library runs with the same permissions as the application it accompanies, whereas the API implementation in the system process has no restrictions. The library provides syntactic sugar for interacting with the API implementation. API calls that read or change global phone state are proxied by the library to the API implementation in the system process.

API calls are handled in three steps (Figure 3.2). First, the application invokes the public API in the library. Second, the library invokes a private interface, also in the library. The private interface is a stub for making Remote Procedure Calls (RPCs). Third, the RPC stub initiates an RPC request that asks a system service to perform the desired operation. For example, if an application calls `ClipboardManager.getText()`, the call will be relayed to `IClipboardStubProxy`, which proxies the call to the system process's `ClipboardService`.

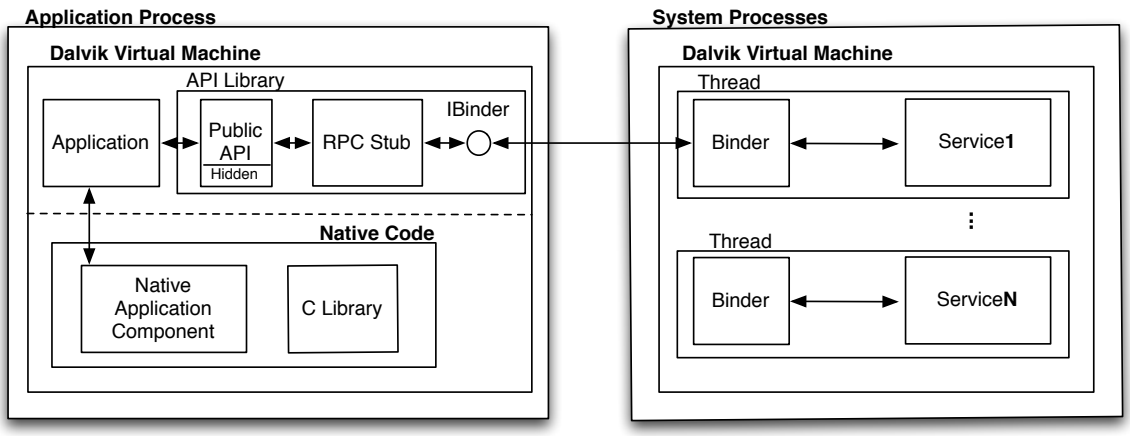


Figure 3.2. The Android platform architecture. Permission checks occur in the system process.

Applications are written in Java, and applications can use Java reflection [109] to access all of the API library's hidden and private classes, methods, and fields. Some private interfaces do not have any corresponding public API; however, applications can still invoke them using reflection. These non-public library methods are intended for use by Google applications or the framework itself, and developers are advised against using them because they may change or disappear between releases [77]. Nonetheless, some applications use them. Java code running in the system process is in a separate virtual machine and therefore immune to reflection.

To enforce permissions, various parts of the system invoke a permission validation mechanism to check whether a given application has a specified permission. The permission validation mechanism is implemented as part of the trusted system process, and invocations of the permission validation mechanism are spread throughout the API. There is no centralized policy for checking permissions when an API is called. Rather, mediation is contingent on the correct placement of permission validation calls.

Permission checks are placed in the API implementation in the system process. When necessary, the API implementation calls the permission validation mechanism to check that the invoking application has the necessary permissions. In some cases, the API library may also redundantly check these permissions, but such checks cannot be relied upon: applications can circumvent them by directly communicating with the system process via the RPC stubs. Permission checks therefore should not occur in the API library. Instead, the API implementation in the system process should invoke the permission validation mechanism.

A small number of permissions are enforced by Linux groups, rather than the Android permission validation mechanism. (We learned about this during permission testing.) In particular, when an application is installed with the `INTERNET`, `WRITE_EXTERNAL_STORAGE`, or `BLUETOOTH` permissions, it is assigned to a Linux group that has access to the pertinent sockets and files. Thus, the kernel enforces the access control policy for these permissions. The API library (which runs with the same rights as the application) can accordingly operate directly on these sockets and files, without needing to invoke the API implementation in the system process.

Applications can include native code in addition to Java code, but native code is still beholden to the permission system. Attempts to open sockets or files are mediated by Linux permissions. It would be very difficult for native code to communicate directly with the system API; the developer would need to re-implement the communication protocol. Instead, applications create Java wrapper methods to invoke the API on behalf of the native code. Android permissions are enforced as usual when the API calls are executed.

Content Providers. System Content Providers are system databases that are separate from the system process and API library. Android uses two ways to restrict access to Content Providers. First, *static* declarations assign separate Android permissions to a given Content Provider's read and write operations when the Content Provider is defined. By default, these permissions are applied to all resources stored by the Content Provider. Restrictions can also be applied at a finer granularity by associating permissions with a path (e.g., `content://a/b`). Second, Content Providers can also enforce permissions *programmatically*: the Content Provider code that handles a query can explicitly call the system's permission validation mechanism.

Intents. To prevent applications from mimicking system Intents, Android restricts who may send certain Intents. All Intents are sent through the ActivityManagerService (a system service), which enforces this restriction. Two techniques are used to restrict the sending of system Intent. Some Intents can only be sent by applications with appropriate permissions. Other system Intents can only be sent by processes whose UID matches the system's. Intents in the latter category cannot be sent by applications, regardless of what permissions they hold, because these Intents must originate from the system process. Applications may also need permissions to receive some system Intents, which the ActivityManagerService checks before delivering the Intents.

3.2 Chrome Extensions

We define Chrome extension terminology and then present Chrome's security features.

3.2.1 Extension Architecture

The Chrome extension platform allows third-party code to run as part of the browser. Extensions change the user's browsing experience by editing web sites[1] and changing browser behavior.

Google Chrome extensions can have three types of components:

- *Content scripts* read and modify websites as needed. They are injected into websites: when the page loads, the content script's JavaScript executes in the context of the site. A content script has full access to its host site's page. Content scripts do not have any HTML or UI elements of their own; instead, they can insert elements into the host page.

[1] When we say *web site*, we refer specifically to the client-side representation of the web site in the browser.

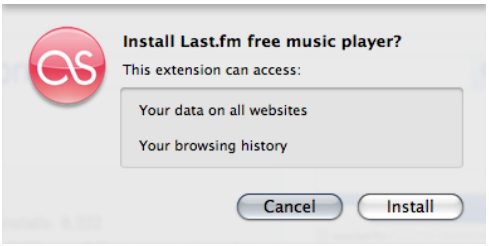


Figure 3.3. Google Chrome extension installation.

- *Core extensions* comprise the main, persistent portions of extensions. They implement features that do not directly involve websites, such as background jobs, preference panels, and other browser UI elements. Core extensions are written in JavaScript and HTML.
- *Plug-ins* are native executables. They can be used to implement complex features. Extensions with plug-ins can only be listed in the official Market if the developer submits the extension for a security review [8].

3.2.2 Chrome Extension Security Model

Many Firefox extensions have publicly suffered from vulnerabilities [104, 154]. To prevent this, the Google Chrome extension platform was designed to protect users from vulnerabilities in benign-but-buggy extensions [31]. It features four security mechanisms: permissions, privilege separation, isolated worlds, and Content Security Policy.

Permissions. By default, extensions cannot use parts of the browser API that impact users' privacy or security. In order to gain access to these APIs, a developer must specify the desired permissions in a file that is packaged with the extension. The extension gallery prompts the user with a warning during installation (e.g., Figure 3.3) that indicates what privileges the extension has requested. An extension is limited to the permissions that its developer requested.

Extensions with plug-ins have the most permissions: plug-ins are native executables, so a plug-in grants the extension full access to the user's machine. The installation warning for an extension with a plug-in says the extension "can access all data on your computer." Extensions can also specify permissions for various browser managers. Each manager is controlled with one permission. For example, an extension must request the `bookmarks` permission to read or alter the user's bookmarks via the bookmark manager. Permissions also restrict extensions' use of content scripts and cross-origin `XMLHttpRequests` (XHRs).[2] Extensions need to specify the domains that it wants to interact with. They can request all-domain access or list specific domains, using wildcards for protocols or subdomains (e.g., `*://*.foo.com`).

[2]Cross-origin XHRs allow extensions to fetch content from any domain, which web sites are not allowed to do [5].

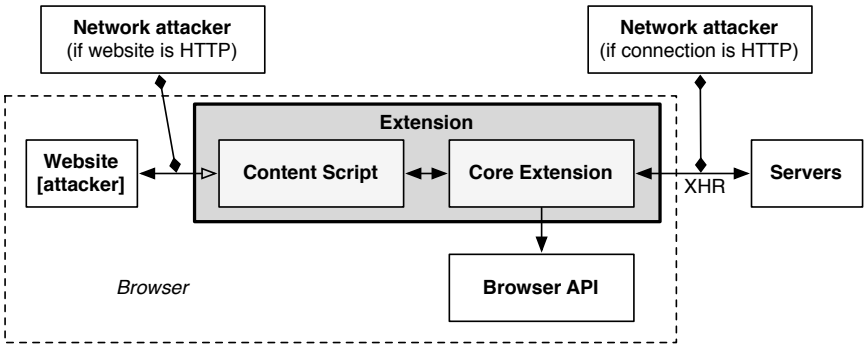


Figure 3.4. The architecture of a Google Chrome extension, annotated with external threats.

Privilege Separation. Chrome extensions are primarily built from content scripts and core extensions, and these components run in separate processes. Content scripts and core extensions communicate by sending JSON messages to each other. Only the core extension can make API calls that require permissions; content scripts cannot invoke browser APIs or make cross-origin XHRs.[3] A content script has only two privileges: it can access the website it is running on, and send messages to its core extension. Figure 3.4 illustrates privilege separation.

The purpose of this architecture is to shield the privileged part of an extension (i.e., the core extension) from attackers. Content scripts are at the highest risk of attack because they directly interact with websites, so they are low-privilege. The sheltered core extension is higher-privilege. As such, an attack that only compromises a content script does not pose a significant threat to the user unless the attack can be extended across the message-passing channel to the higher-privilege core extension. (Binary plug-ins also run in their own processes, but privilege separation is not used to shield them. Instead, they undergo a manual security review process.)

Isolated Worlds. The isolated worlds mechanism is intended to protect content scripts from web attackers. A content script can read or modify a website's DOM, but the content script and website have separate JavaScript heaps with their own DOM objects. Consequently, content scripts and websites never exchange pointers.[4]

Content Security Policy. Contemporaneously with our case study (Chapter 5), the Google Chrome extension team began to implement Content Security Policy (CSP) for extensions. CSP allows an extension developer to specify a policy that governs what type of content can be included in the website [138]. CSP can restrict the domains that scripts, frames, and images come from. Additionally, a policy can specify whether inline scripts and `eval` are allowed to execute in the page context. If used correctly, CSP can prevent cross-site scripting and other code injection attacks. For example, untrusted data that is accidentally inserted into HTML cannot execute if the extension's policy disallows inline scripts.

[3] In newer versions of Chrome, content scripts can make cross-origin XHRs [5]. However, this was not permitted at the time of our study (Chapter 5).

[4] Although isolated worlds separates websites from content scripts, it not a form of privilege separation; privilege separation refers to techniques that isolate parts of the same application from each other [122, 127].

Chapter 4

Developers' Permission Request Patterns

4.1 Introduction

Application permissions offer two advantages over traditional user-based permissions: defense-in-depth and user consent. However, these benefits rely on the assumption that applications generally request fewer than full privileges. One challenge is that developers might routinely assign more permissions to their applications than they require, due to confusion or indifference. We explore whether the assumption of a low permission request rate is realized in practice, by performing large-scale studies of Android applications and Google Chrome extensions.

We collect the permission requirements of 956 Android applications and 1,000 Google Chrome extensions. From this data, we measure the overall rate at which developers request permissions and discuss the effects of the rate on defense-in-depth and user consent. We then evaluate the rates of overprivilege in Android applications and Google Chrome extensions by analyzing the applications and extensions; overprivileged applications unnecessarily increase the impact of bugs and expose users to extra permission warnings.

Our results indicate that application permissions can benefit system security. We find that almost all applications ask for fewer than maximum permissions: only 14 of 1,000 extensions request the most dangerous privileges, and the average Android application requests fewer than 4 of 56 available dangerous permissions. These results indicate that application permission systems with up-front permission declarations have the potential to decrease the impact of application vulnerabilities and simplify review. We further explore this in Chapter 5 by examining the impact of permissions on actual vulnerabilities in applications.

Although developers use fewer than full permissions, we find evidence of overprivilege. Our review of privilege usage found that about one-third of Android applications and fifteen percent of Chrome extensions are overprivileged. We investigate causes of overprivilege and find that many instances of overprivilege stem from confusion about the permission systems. Our results indicate

that developers are trying to follow least privilege, which supports the potential effectiveness of permissions. Consequently, developer tools could further decrease the permission request rate.

Unfortunately, the permission request rate may not be low enough to achieve the goal of user consent. Our measurements show that Google Chrome and Android users are presented with at least one dangerous permission request during the installation of almost every extension and application. Warning science literature indicates that frequent warnings desensitize users, especially if most warnings do not lead to negative consequences [139, 107]. We therefore hypothesize that users are not likely to pay attention to or gain information from the install-time permission prompts in these systems; we further explore this hypothesis in Chapter 6.

4.2 Permission Request Rates

We examine the frequency of permission requests. These results should be compared to traditional systems, which grant all applications full privileges.

4.2.1 Android Applications

We survey 100 paid and 856 free applications from the Android Market, collected in October 2010. For the paid applications, we selected the 100 most popular. The free set is comprised of the 756 most popular and 100 most recently added applications. Although Android applications written by the same developer could theoretically collude, we consider each application's permissions independently. Legitimate developers have no incentive to circumvent permission warnings.

Dangerous Permissions

We are primarily concerned with the prevalence of "Dangerous" permissions, which are displayed as warnings to users during installation and can have serious security ramifications if abused. We find that 93% of free and 82% of paid applications have at least one Dangerous permission.

Although many applications ask for at least one Dangerous permission, the number of permissions required by an application is typically low. Even the most highly privileged application in our set asks for less than half of the available 56 Dangerous permissions. Figure 4.1 shows the distribution of Dangerous permission requests. Paid applications use an average of 3.99 Dangerous permissions, and free applications use an average of 3.46 Dangerous permissions.

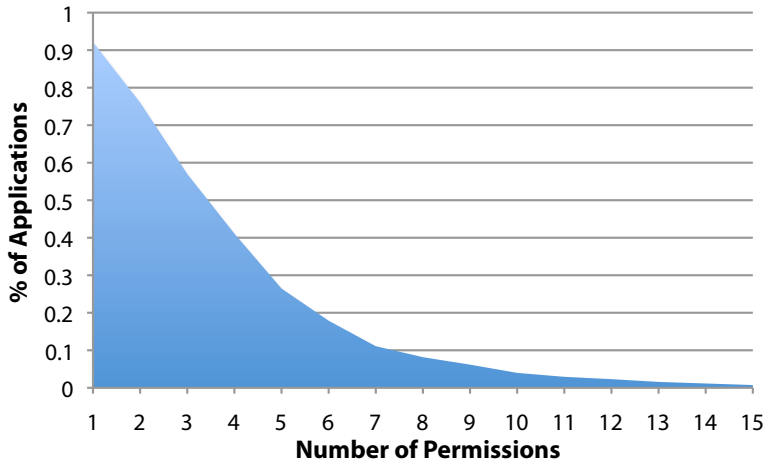


Figure 4.1. Percentage of paid and free applications with *at least* the given number of Dangerous permissions.

A small number of permissions are requested very frequently. Table 4.1(a) shows the most popular Dangerous permissions. In particular, the `INTERNET` permission is heavily used. We find that 14% of free and 4% of paid applications request `INTERNET` as their only Dangerous permission. Barrera et al. hypothesize that free applications often need the `INTERNET` permission only to load advertisements [30]. The disparity in `INTERNET` use between free and paid applications supports this hypothesis, although it is still the most popular permission for paid applications.

The prevalence of the `INTERNET` permission means that most applications with access to personal information also have the ability to leak it. For example, 97% of the 225 applications that ask for `ACCESS_FINE_LOCATION` also request the `INTERNET` permission. Similarly, 94% and 78% of the respective applications that request `READ_CONTACTS` and `READ_CALENDAR` also have the `INTERNET` permission. We find that significantly more free than paid applications request both Internet access and location data, which possibly indicates widespread leakage of location information to advertisers in free applications. This corroborates a previous study that found that 20 of 30 randomly selected free applications send user information to content or advertisement servers [62].

Android permissions are grouped into functionality categories, and Table 4.1 shows how many applications use at least one Dangerous permission from each given category. This provides an overview of the types of permissions that applications request. During installation, all of the permissions in a category are grouped together into one warning under the category's heading, so Table 4.1 also indicates how often users see each type of warning.

Signature and System Permissions

Applications can request Signature and SignatureOrSystem permissions, but the operating system will not grant the request unless the application has been signed by the device manufacturer (Signature) or installed in the `/system/app` folder (System). It is pointless for a typical application to request these permissions because the permission requests will be ignored.

Permission (Category)	Free	Paid
INTERNET** (NETWORK)	86.6 %	65 %
WRITE_EXTERNAL_STORAGE** (STORAGE)	34.1 %	50 %
ACCESS_COARSE_LOCATION** (LOCATION)	33.4 %	20 %
READ_PHONE_STATE (PHONE_CALLS)	32.1 %	35 %
WAKE_LOCK** (SYSTEM_TOOLS)	24.2 %	40 %
ACCESS_FINE_LOCATION (LOCATION)	23.4 %	24 %
READ_CONTACTS (PERSONAL_INFO)	16.1 %	11 %
WRITE_SETTINGS (SYSTEM_TOOLS)	13.4 %	18 %
GET_TASKS* (SYSTEM_TOOLS)	4.4 %	11 %

(a) The most frequent Dangerous permissions and their categories.

Category	Free	Paid
NETWORK**	87.3 %	66 %
SYSTEM_TOOLS	39.7 %	50 %
STORAGE**	34.1 %	50 %
LOCATION**	38.9 %	25 %
PHONE_CALLS	32.5 %	35 %
PERSONAL_INFO	18.4 %	13 %
HARDWARE_CONTROLS	12.5 %	17 %
COST_MONEY	10.6 %	9 %
MESSAGES	3.7 %	5 %
ACCOUNTS	2.6 %	2 %
DEVELOPMENT_TOOLS	0.35 %	0 %

(b) Prevalence of each permission category.

Table 4.1. Survey of the permissions used by 856 free and 100 paid Android applications. Android documentation defines the permissions listed here [7]. We indicate significant differences between the free and paid applications at 1% (**) and 5% (*) significance levels, computed with z-tests.

Eight paid applications request Signature/System permissions, but they are unlikely to receive these permissions: to our knowledge, none of these applications are signed by device manufacturers. Of the free applications, 25 request Signature permissions, 30 request SignatureOrSystem permissions, and four request both. We have found four of these free applications pre-installed on phones; the remainder will not receive the permissions on a typical device. Requests for unobtainable permissions may be developer error or left over from testing.

Implications

Defense in Depth. Given the prevalence of Dangerous permissions, an application vulnerability is likely to occur in an application with at least one Dangerous permission. However, the average Android application is much less privileged than a traditional operating system program. Every desktop Windows application has full privileges, whereas no Android application in our set requests more than half of the available Dangerous permissions. A majority of the Android applications ask for less than seven, and only 10% have access to functionality that costs the user money. This is a significant improvement over the traditional full-privilege, user-based approach.

User Consent. Nearly all applications (93% of free and 82% of paid) ask for at least one Dangerous permission, which indicates that users often install applications with Dangerous permissions. We hypothesize that this high request rate causes users to become habituated to clicking through Android's permission warnings. The `INTERNET` permission is so widely requested that users cannot consider its warning anomalous. Security guidelines or anti-virus programs that warn against installing applications with access to both the Internet and personal information are likely to fail because almost all applications with personal information also have `INTERNET`.

4.2.2 Chrome Extensions

We study the 1,000 "most popular" extensions, as ranked in the official Google Chrome extension gallery[1]. Of these, the 500 most popular extensions are relevant to user consent and application vulnerabilities because they comprise the majority of user downloads. The 500 less popular extensions are installed in very few browsers. Table 4.2 provides an overview of our results.

Popular Extensions

Of the 500 most popular extensions, 91.4% ask for at least one security-relevant permission. This indicates that nearly every installation of an extension generates at least one security warning.[2]

[1] We crawled the directory on August 27, 2010.

[2] We discovered that Google Chrome sometimes fails to generate a warning for history access. The bug has been fixed for new versions [66]. Our analysis assumes that all requests for history access correctly generate a warning. The bug affects 5 of extensions in our set.

Permission	Popular	Unpopular
Plug-ins	2.80 %	0.00 %
Web access	82.0 %	60.8 %
All domains	51.6 %	21.8 %
Specific domains	30.4 %	39.0 %
Browser manager(s)	74.8 %	43.4 %

Table 4.2. We measure the prevalence of permissions in 1,000 Google Chrome extensions, split into the 500 most popular and 500 less popular. For web access, we report the highest permission of either the content script or core extension.

Plug-ins. Only 14 of the 500 extensions include plug-ins.

Browser Managers. The majority of security warnings are caused by the window manager, which is requested by almost 75% of the 500 extensions. Requesting access to the window manager generates a warning about history access because history is available through the window manager. The bookmark and geolocation managers are requested infrequently: 44 times and once, respectively.

All Domains. Half of the 500 extensions request all-domain access for either content scripts or the core extension. 52% request access to all `http` sites, and 42% ask for all `https` sites.

Specific Domains. One-third of extensions only request a set of specific domains. This reduces the attack surface and reduces the possibility that an extension is snooping on sensitive web data; for example, none of the extensions request specific domain access to financial web sites.

No Warning. Only 43 of the 500 extensions do not request access to a security-relevant permission. They load normal web sites into their extension windows, apply "themes" to the user interface, or use browser managers that are not relevant to privacy or security.

Unpopular Extensions

Not all of the extensions listed in the "most popular" directory ranking are popular (Figure 4.2). After approximately the first 500 of 1,000 popularity-ranked extensions, the number of users per extension abruptly decreases, and applications are no longer ranked solely according to the number of users. (Although the ranking algorithm is private, we believe it incorporates factors beyond the number of downloads such as posting dates and ratings.) 16.2% of the bottom 500 extensions have fewer than ten users. These 500 low-ranked extensions are of uneven quality. E.g., two of them are unaltered copies of the example extension on the developer web site.

71.6% of the less popular extensions have at least one security-relevant permission. Table 4.2 breaks down the permission usage of the 500 less popular extensions. When compared to the top 500 extensions, the unpopular extensions request far fewer permissions than popular extensions. All of the differences are significant at a 1% significance level. We hypothesize that this is because less popular extensions offer less functionality.

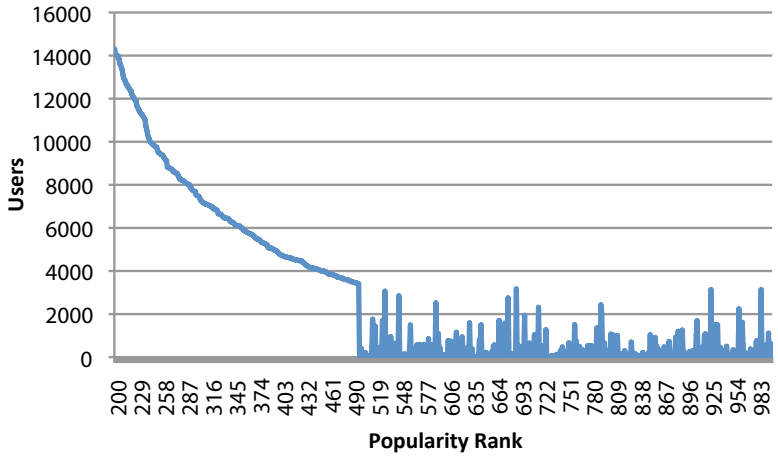


Figure 4.2. Users per extension, sorted by the official gallery's ranking. We omit the first 200 for graph clarity; on the date of our survey, the most popular extension had 1.3M users.

Unranked extensions are as unpopular as the unpopular extensions in our data set. If one were to review the remaining 5,696 unranked Google Chrome extensions, we expect their permission requirements would be equivalent to or less than the permission requirements of these 500 unpopular applications. We note with caution that future studies on permissions need to consider the effect of popularity: a study that were to look at the full set of 6,696 extensions to evaluate warning frequency would underestimate the number of warnings that users see in practice by approximately 20% because installations are heavily concentrated in the most popular extensions. However, the effect of popularity may be smaller in other application markets; for example, we did not observe a relationship between permissions and popularity in Android applications.

Evaluation

Defense in Depth. This study shows that the permission system dramatically reduces the scope of potential extension vulnerabilities. A negligible number of extensions include plug-ins, which means that the typical extension vulnerability cannot yield access to the local machine. This is a significant improvement over the Firefox and Internet Explorer extension systems, which provide *all* extensions with access to the local file system. We also find that all-domain access is frequent but not universal: 18% of popular extensions need no web access, and 30.4% only need limited web access. This means that the permission system prevents *half* of popular extensions from having unnecessary web privileges (e.g., access to financial information).

User Consent. Nearly all popular extensions (91% of the top 500) generate at least one security warning, which decreases the value of the warnings. History and all-domain permissions are requested by more than half of extensions; consequently, users may have grown accustomed to them. However, warnings about plug-ins are rare and therefore potentially notable.

4.3 Overprivilege in Android Applications

We studied Android applications to determine whether developers follow least privilege with their permission requests. We built Stowaway, a tool that detects overprivilege in compiled Android applications. Stowaway is composed of two parts: a static analysis tool that determines what API calls an application makes (Chapter 4.3.1), and a permission map that identifies what permissions are needed for each API call. Android's documentation does not provide a permission map, so we empirically determined Android 2.2's access control policy (Chapters 4.3.2 and 4.3.3). We apply Stowaway to a set of 940 applications and find that about one-third are overprivileged (Chapter 4.3.4). We investigate the causes of overprivilege and find evidence that developers are trying to follow least privilege but sometimes fail due to insufficient API documentation. Better documentation and developer tools could help reduce the Android permission request rate.

4.3.1 Application Analysis Tool

We built a static analysis tool, Stowaway, which analyzes an Android application and determines the maximum set of permissions it may require. Stowaway takes compiled Android applications as input. Compiled applications for the Android platform are in Dalvik executable (DEX) format [143]. The DEX files are bundled with XML files that describe application layout. Stowaway extracts XML files from the target applications' package and uses Dedexer [120] to disassemble the target application's code. Each subsequent stage uses the disassembled DEX and XML files as input. Stowaway analyzes the application's use of API calls, Content Providers, and Intents and then uses the permission map built in Chapter 4.3.2 to determine what permissions they require.

API Calls

Stowaway first parses the disassembled DEX files and identifies all calls to standard API methods. Stowaway tracks application-defined classes that inherit methods from Android classes so we can differentiate between invocations of application-defined methods and Android-defined inherited methods. We use heuristics to handle Java reflection and two unusual permissions. Our static analysis is primarily linear; we do not handle all non-linear control flows, such as `jumps`. We only handle simple `gotos` that appear at the ends of methods.

Reflection. Java reflection is a challenging problem [105, 39, 130]. In Java, methods can be reflectively invoked with `java.lang.reflect.Method.invoke()` or `java.lang.reflect.Constructor.newInstance()`. Stowaway tracks which Class objects and method names are propagated to the reflective invocation. It performs flow-sensitive, intra-procedural static analysis, augmented with inter-procedural analysis to a depth of two method calls. Within each method body, it statically tracks the value of each String, StringBuilder, Class, Method, Constructor, Field, and Object. We also statically track the state of static member vari-

ables of these types. We identify method calls that convert strings and objects to type Class, as well as method calls that convert Class objects to Methods, Constructors, and Fields.

We also apply Android-specific heuristics to resolving reflection by handling methods and fields that may affect reflective calls. We cannot model the behavior of the entire Android and Java APIs, but we identify special cases. First, `Context.getSystemService(String)` returns different types of objects depending on the argument. We maintain a mapping of arguments to the types of return objects. Second, some API classes contain private member variables that hold references to hidden interfaces. Applications can only access these member variables reflectively, which obscures their type information because they are treated like generic `Objects`. We created a mapping between member variables and their types and propagate the type data accordingly. If an application subsequently accesses methods on a member variable after retrieving it, we can resolve the member variable's type.

Internet. Any application that includes a *WebView* must have the Internet permission. A WebView is a user interface component that allows an application to embed a web site into its UI. WebViews can be instantiated programmatically or declared in XML files. Stowaway identifies programmatic instantiations of WebViews. It also decompiles application XML files and parses them to detect WebView declarations.

External Storage. If an application wants to access files stored on the SD card, it must have the `WRITE_EXTERNAL_STORAGE` permission. This permission does not appear in our permission map because it (1) is enforced entirely using Linux permissions and (2) can be associated with any file operation or API call that accesses the SD card from within the library. We handle this permission by searching the application's string literals and XML files for strings that contain `sdcard`; if any are found, we assume `WRITE_EXTERNAL_STORAGE` is needed. Additionally, we assume this permission is needed if we see API calls that return paths to the SD card, such as `Environment.getExternalStorageDirectory()`.

Content Providers

Content Providers are accessed by performing a database operation on a URI. Stowaway collects all strings that could be used as Content Provider URIs and links those strings to the Content Providers' permission requirements. Content Provider URIs can be obtained in two ways:

1. A string or set of strings can be passed into a method that returns a URI. For example, the API call `android.net.Uri.parse("content://browser/bookmarks")` returns a URI for accessing the Browser bookmarks. To handle this case, Stowaway finds all string literals that begin with `content://`.
2. The API provides Content Provider helper classes that include public URI constants. For example, the value of `android.provider.Browser.BOOKMARKS_URI` is `content://browser/bookmarks`. Stowaway recognizes known URI constants, and we created a mapping from all known URI constants to their string values.

A limitation of our tool is that we cannot tell which database operations an application performs with a URI; there are many ways to perform an operation on a Content Provider, and users can set

their own query strings. To account for this, we say that an application may require any permission associated with any operation on a given Content Provider URI. This provides an upper bound on the permissions that could be required in order to use a specific Content Provider.

Intents

We use ComDroid [47] to detect the sending and receiving of Intents that require permissions. ComDroid performs flow-sensitive, intra-procedural static analysis, augmented with limited inter-procedural analysis that follows method invocations to a depth of one method call. ComDroid tracks the state of Intents, registers, Intent-sending sinks (e.g., `sendBroadcast`), and application components. When an Intent object is instantiated, passed as a method parameter, or obtained as a return value, ComDroid statically tracks all changes to it from its source to its sink and outputs all information about the Intent and all components expecting to receive messages.

Stowaway takes ComDroid's output and, for each sent Intent, checks whether a permission is required to send that Intent. For each Intent that an application is registered to receive, Stowaway checks whether a permission is required to receive the Intent. Occasionally ComDroid is unable to identify the message or sink of an Intent. To mitigate these cases, Stowaway searches for protected system-wide Intents in the list of all string literals in the application.

4.3.2 Permission Testing Methodology

Android's access control policy is not well documented, but the policy is necessary to determine whether applications are overprivileged. To address this shortcoming, we empirically determined the access control policy that Android enforces. We used testing to construct a permission map that identifies the permissions required for each method in the Android API. In particular, we modified Android 2.2's permission verification mechanism to log permission checks as they occur. We then generated unit test cases for API calls, Content Providers, and Intents. Executing these tests allowed us to observe the permissions required to interact with system APIs. A core challenge was to build unit tests that obtain call coverage of all platform resources. The resulting permission map is publicly available at `http://www.android-permissions.org`.

We chose to use dynamic testing to generate the permission map because of the complexities of Android's architecture (Chapter 3.1.4). Android's API is written in two languages (Java and C++), crosses multiple processes, and includes code that is only reachable under factory conditions (i.e., when certain pre-consumer variables are set to be `true`). However, in concurrent work, Gibler et al. [72] applied static analysis to the Android API to find permission checks. Unlike our dynamic approach, their static analysis might have false positives and will miss permission checks in native code.

The API

As described in Chapter 3.1.4, the Android API provides applications with a library that includes public, private, and hidden classes and methods. The set of private classes includes the RPC stubs for the system services.[3] All of these classes and methods are accessible to applications using Java reflection, so we must test them to identify permission checks. We conducted testing in three phases: feedback-directed testing; customizable test case generation; and manual verification.

Feedback-Directed Testing. For the first phase of testing, we used Randoop, an automated, feedback-directed, object-oriented test generator for Java [118, 119].[4] Randoop takes a list of classes as input and tries different orderings of the methods from these classes. We modified Randoop to run as an Android application and to log every method it invokes. Our modifications to Android log every permission that is checked by the Android permission validation mechanism, which lets us deduce which API calls trigger permission checks.

Randoop maintains a pool of valid initial input sequences and parameters, initially seeded with primitive values (e.g., `int` and `String`). Randoop builds test sequences incrementally by randomly selecting a method from the test class's methods and selecting sequences from the input pool to populate the method's arguments. If the new sequence is unique, then it is executed. Sequences that complete successfully (i.e., without generating an exception) are added to the sequence pool. Randoop's goal is full coverage of the test space. Unlike comparable techniques [117, 52, 26], Randoop does not need a sample execution trace as input, making large-scale testing such as API fuzzing more manageable. Because Randoop uses Java reflection to generate the test methods from the supplied list of classes, it supports testing non-public methods. We modified Randoop to also test nested classes of the input classes.

Randoop's feedback-guided space exploration is limited by the objects and input values it has access to. If Randoop cannot find an object of the correct type needed to invoke a method in the sequence pool, then it will never try to invoke the method. The Android API is too large to test all interdependent classes at once, so in practice many objects are not available in the sequence pool. We mitigated this problem by testing related classes together (for example, `Account` and `AccountManager`) and adding seed sequences that return common Android-specific data types. Unfortunately, this was insufficient to produce valid input parameters for many methods. Many singleton object instances can only be created through API calls with specific parameters; for example, a `WifiManager` instance can be obtained by calling `android.content.Context.getSystemService(String)` with the parameter `"wifi"`. We addressed this by augmenting the input pool with specific primitive constants

[3] The operating system also includes many internal methods that make permission checks. However, applications cannot invoke them because they are not currently exposed with RPC stubs. Since we are focused on the application-facing API, we do not test or discuss these permission checks.

[4] Randoop is not the only Java unit test generation tool. Tools like Eclat [117], Palulu [26] and JCrasher [52] work similarly but require an example execution as input. Given the size of the Android API, building such an example execution would be a challenge. Enhanced JUnit [63] generates tests by chaining constructors to some fixed depth. However, it does not use subtyping to provide instances and relies on bytecode as input. Korat [43] requires formal specifications of methods as input, which is infeasible for post-facto testing of the Android API.

and sequences. Additionally, some API calls expect memory addresses that store specific values for parameters, which we were unable to solve at scale.

Randoop also does not handle ordering requirements that are independent of input parameters. In some cases, Android expects methods to precede each other in a very specific order. Randoop only generates sequence chains for the purpose of creating arguments for methods; it is not able to generate sequences to satisfy dependencies that are not in the form of an input variable. Further aggravating this problem, many Android methods with underlying native code generate segmentation faults if called out of order, which terminates the Randoop testing process.

Customizable Test Case Generation. Randoop's feedback-directed approach to testing failed to cover certain types of methods. When this happened, there was no way to manually edit its test sequences to control sequence order or establish method pre-conditions. To address these limitations and improve coverage, we built our own test generation tool. Our tool accepts a list of method signatures as input, and outputs at least one unit test for each method. It maintains a pool of default input parameters that can be passed to methods to be called. If multiple values are available for a parameter, then our tool creates multiple unit tests for that method. (Tests are created combinatorially when multiple parameters of the same method have multiple possible values.) It also generates tests using null values if it cannot find a suitable parameter. Because our tool separates test case generation from execution, a human tester can edit the test sequences produced by our tool. When tests fail, we manually adjust the order of method calls, introduce extra code to satisfy method pre-conditions, or add new parameters for the failing tests.

Our test generation tool requires more human effort than Randoop, but it is effective for quickly achieving coverage of methods that Randoop was unable to properly invoke. Overseeing and editing a set of generated test cases produced by our tool is still substantially less work than manually writing test cases. Our experience with large-scale API testing was that methods that are challenging to invoke by feedback-directed testing occur often enough to be problematic. When a human tester has the ability to edit failing sequences, these methods can be properly invoked.

Manual Verification. The first two phases of testing generate a map of the permission checks performed by each method in the API. However, these results contain three types of inconsistencies. First, the permission checks caused by asynchronous API calls are sometimes incorrectly associated with subsequent API calls. (For example, one of the methods to connect to a network returns immediately; the system service then continues to set up the connection.) Second, a method's permission requirements can be argument-dependent, in which case we see intermittent or different permission checks for that method. Third, permission checks can be dependent on the order in which API calls are made. To identify and resolve these inconsistencies, we manually verified the correctness of the permission map generated by the first two phases of testing.

We used our customizable test generation tool to create tests to confirm the permission(s) associated with each API method in our permission map. We carefully experimented with the ordering and arguments of the test cases to ensure that we correctly matched permission checks to asynchronous API calls and identified the conditions of permission checks. When confirming permissions for potentially asynchronous or order-dependent API calls, we also created confirmation test cases for related methods in the pertinent class that were not initially associated with permission checks. We ran every test case both with and without their required permissions in order

to identify API calls with multiple or substitutable permission requirements. (For example, the method `killBackgroundProcesses` accepts either the `RESTART_PACKAGES` permission or the `KILL_BACKGROUND_PROCESSES` permission.) If a test case throws a security exception without a permission but succeeds with a permission, then we know that the permission map for the method under test is correct.

Testing The Internet Permission. Applications can access the Internet through the Android API, but other packages such as `java.net` and `org.apache` also provide Internet access. In order to determine which methods require access to the Internet, we scoured the documentation and searched the Internet for any and all methods that suggest Internet access. Using this list, we wrote test cases to determine which of those methods require the `INTERNET` permission.

Content Providers

Our Content Provider test application executes `query`, `insert`, `update`, and `delete` operations on Content Provider URIs associated with system Content Providers. We collected a list of URIs from the `android.provider` package to determine the core set of Content Providers to test. We additionally collected Content Provider URIs that we discovered during other phases of testing. For each URI, we attempted to execute each type of database operation without any permissions. If a security exception was thrown, we recorded the required permission. We added and tested combinations of permissions to identify multiple or substitutable permission requirements. Each Content Provider was tested until security exceptions were no longer thrown for a given operation, indicating the minimum set of permissions required to complete that operation. In addition to testing, we also examined the system Content Providers' static permission declarations.

Intents

We built a pair of applications to send and receive Intents. The Android documentation does not provide a single, comprehensive list of the available system Intents, so we scraped the Android 2.2 source code to find string constants that could be the contents of an Intent.[5] For example, the string `android.net.wifi.STATE_CHANGE` is used to send or receive the system-wide Intent that notifies applications about an Internet connectivity change. We sent and received Intents with these constants between our test applications. In order to test the permissions needed to receive system broadcast Intents, we triggered system broadcasts by sending and receiving text messages, sending and receiving phone calls, connecting and disconnecting WiFi, connecting and disconnecting Bluetooth devices, and other actions. For all of these tests, we recorded whether permission checks occurred and whether the Intents were delivered or received successfully.

[5]For those familiar with Android terminology, we searched for Intent *action* strings [6].

4.3.3 Permission Map Results

Our testing of the Android application platform resulted in a permission map that correlates permission requirements with API calls, Content Providers, and Intents. In this section, we discuss our coverage of the API, compare our results to the official Android documentation, and present characteristics of the Android API and permission map.

Coverage

The Android 2.2 API consists of 1,665 classes with a total of 16,732 public and private methods. We covered 85% the methods through two phases of testing. (We define a method as *covered* if we executed it without generating an exception; we do not measure branch coverage.) Randoop attained an initial method coverage of 60%, spread across all packages. We supplemented Randoop's coverage with our proprietary test generation tool, accomplishing close to 100% coverage of the classes with at least one permission check (as per our first round of testing).

The uncovered portion of the API is due to native calls and the omission of second-phase tests for packages that did not yield permission checks in the first phase. First, native methods often crashed the application when incorrect parameters were supplied, making them difficult to test. Many native method parameters are integers that represent pointers to objects in native code, making it difficult to supply correct parameters. Approximately one-third of uncovered methods are native calls. Second, we decided to omit supplemental tests for packages that did not reveal permission checks during the Randoop testing phase. If Randoop did not trigger at least one permission check in a package, we did not add more tests to the classes in the package.

We investigated a total of 62 Content Providers. We found that there are 18 Content Providers that do not have permissions for insert, query, update, or delete. All of the Content Providers that lack permissions are associated with the `media` content URI.

We examined Intent communication and measured whether permissions are required for sending and receiving Intents. When sending broadcast Intents, 62 broadcasts are prohibited by non-system senders, 6 require permissions before sending the Intent, and 2 can be broadcast but not received by system receivers. Broadcast receivers must have permissions to receive 23 broadcast Intents, of which 14 are protected by a Bluetooth permission. When sending Intents to start Activities, 7 Intent messages require permissions. When starting Services, 2 Intents require permissions.

Comparison With Documentation

Clear and well-developed documentation promotes correct permission usage and safe programming practices. Errors and omissions in the documentation can lead to incorrect developer assumptions and overprivilege. Android's documentation of permissions is limited, which is likely due to their lack of a centralized access control policy. Our testing identified 1,259 API calls with permission checks. We compare this to the Android 2.2 documentation, as follows.

Permission	Usage
BLUETOOTH	85
BlUETOOTH_ADMIN	45
READ_CONTACTS	38
ACCESS_NETWORK_STATE	24
WAKE_LOCK	24
ACCESS_FINE_LOCATION	22
WRITE_SETTINGS	21
MODIFY_AUDIO_SETTINGS	21
ACCESS_COARSE_LOCATION	18
CHANGE_WIFI_STATE	16

Table 4.3. Android's 10 most checked permissions; brief descriptions of the permissions are available in the documentation [7].

We crawled the Android 2.2 documentation and found that it specifies permission requirements for 78 methods. The documentation additionally lists permissions in several class descriptions, but it is not clear which methods of the classes require the stated permissions. Of the 78 permission-protected API calls in the documentation, our testing indicates that the documentation for 6 API calls is incorrect. It is unknown to us whether the documentation or implementation is wrong; if the documentation is correct, then these discrepancies may be security errors.

Three of the documentation errors list a different permission than was found through testing. In one place, the documentation claims an API call is protected by the Dangerous permission MANAGE_ACCOUNTS, when it actually can be accessed with the lower-privilege Normal permission GET_ACCOUNTS. Another error claims an API call requires the ACCESS_COARSE_UPDATES permission, which does not exist. As a result, 5 of the 900 applications that we study in Chapter 4.3.4 request this non-existent permission. (Applications can request non-existent permissions; an unobservant developer might miss the errors produced on the console when these applications are loaded.) A third error states that a method is protected with the BLUETOOTH permission, when the method is in fact protected with BLUETOOTH_ADMIN.

The other three documentation errors pertain to methods with multiple permission requirements. In one error, the documentation claims that a method requires one permission, but our testing shows that two are required. For the last two errors, the documentation states that two methods require one permission each; in practice, however, the two methods both accept two permissions.

Characterizing Permissions

Based on our permission map, we characterize how permission checks are distributed.

Number of Permissions Checks. We identified 1,244 API calls with permission checks, which is 6.45% of all API methods (including hidden and private methods). Of those, 816 are methods of normal API classes, and 428 are methods of RPC stubs that are used to communicate with system services. We additionally identified 15 API calls with permission checks in a supplementary part of

the API added by a manufacturer, for a total of 1,259 API calls with permission checks. Table 4.3 provides the rates of the most commonly-checked permissions for the normal API.

Signature/System Permissions. We found that 12% of the non-RPC API calls with permissions are protected with Signature/System permissions, and 35% of the RPC stubs with permissions are protected with Signature/System permissions. This effectively limits the use of these API calls to pre-installed applications.

Unused Permissions. We found that some permissions are defined by the platform but never used within the API. For example, the `BRICK` permission is never used, despite being oft-cited as an example of a particularly dire permission [146]. The only use of the `BRICK` permission is in dead code that is incapable of causing harm to the device. Our testing found that 15 of the 134 Android-defined permissions are unused. For each case where a permission was never found during testing, we searched the source tree to verify that the permission is not used. After examining several devices, we discovered that one of the otherwise unused permissions is used by the custom classes that HTC and Samsung added to the API to support 4G on their phones.

Hierarchical Permissions. The names of many permissions imply that there are hierarchical relationships between them. Intuitively, we expect that more powerful permissions should be substitutable for lesser permissions relating to the same resource. However, we find no evidence of planned hierarchy. Our testing indicates that `BLUETOOTH_ADMIN` is not substitutable for `BLUETOOTH`, nor is `WRITE_CONTACTS` substitutable for `READ_CONTACTS`. Similarly, `CHANGE_WIFI_STATE` cannot be used in place of `ACCESS_WIFI_STATE`.

Only one pair of permissions has a hierarchical relationship: `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`. Every method that accepts the `COARSE` permission also accepts `FINE` as a substitute. We found only one exception to this, which may be a bug: `TelephonyManager.listen()` accepts either `ACCESS_COARSE_LOCATION` or `READ_PHONE_STATE`, but it does not accept `ACCESS_FINE_LOCATION`.

Permission Granularity. If a single permission is applied to a diverse set of functionality, applications that request the permission for a subset of the functionality will have unnecessary access to the rest. Android aims to prevent this by splitting functionality into multiple permissions when possible. As a case study, we examine the division of Bluetooth functionality, as the Bluetooth permissions are the most heavily checked permissions. We find that the two Bluetooth permissions are applied to 6 large classes. They are divided between methods that change state (`BLUETOOTH_ADMIN`) and methods that get device information (`BLUETOOTH`). The `BluetoothAdapter` class is one of several that use the Bluetooth permissions, and it appropriately divides most of its permission assignments. However, it features some inconsistencies. One method only returns information but requires the `BLUETOOTH_ADMIN` permission, and another method changes state but requires both permissions. This type of inconsistency may lead to developer confusion about which permissions are required for which types of operations.

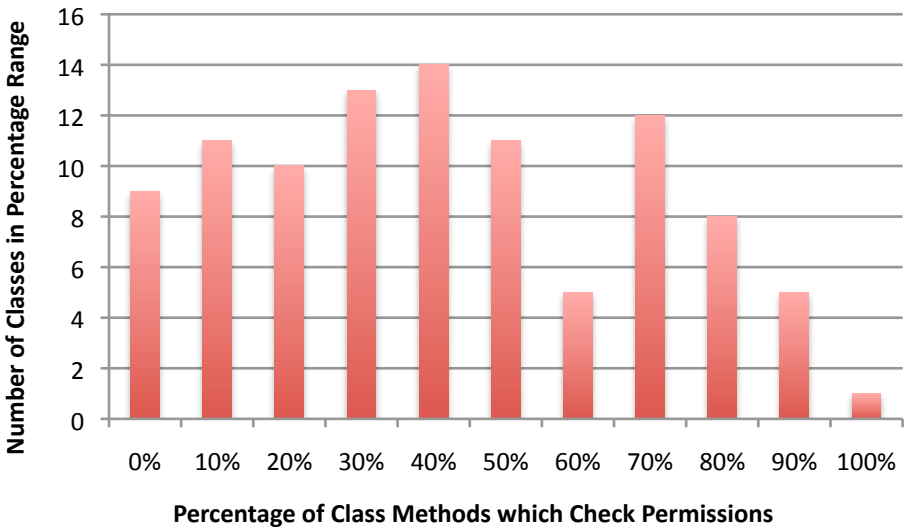


Figure 4.3. A histogram of the number of classes, sorted by the percentage of the classes' methods that require permissions. The numbers shown represent ranges, i.e., 10% represents [10-20%). We only consider classes with at least 1 permission check.

Class Characteristics. Figure 4.3 presents the percentage of methods that are protected per class. We initially expected that the distribution would be bimodal, with most classes protected entirely or not at all. Instead, we see a wide array of class protection rates. Of these classes, 8 require permissions to instantiate an object, and 4 require permissions only for the object constructor.

4.3.4 Application Analysis Results

We applied Stowaway to 940 Android applications to identify the prevalence of overprivilege. Stowaway calculates the maximum set of Android permissions that an application may need. We compare that set to the permissions actually requested by the application. If the application requests more permissions, then it is overprivileged. Our full set of applications is the same as the set discussed in Chapter 4.2.1, although we removed randomly selected applications for tool testing and training. This left us with 940 applications for analysis.

Manual Analysis

Methodology. We randomly selected 40 applications from the set of 940 and ran Stowaway on them. Stowaway identified 18 applications as overprivileged. We then manually analyzed each overprivilege warning to attribute it to either tool error (i.e., a false positive) or developer error. We looked for false positives due to three types of failures:

1. Stowaway misses an API, Content Provider, or Intent operation that needs a permission. For example, Stowaway misses an API call when it cannot resolve the target of a reflective call.
2. Stowaway correctly identifies the API, Content Provider, or Intent operation, but our permission map lacks an entry for that platform resource.
3. The application sends an Intent to some other application, and the recipient accepts Intents only from senders with a certain permission. Stowaway cannot detect this case because we do not determine the permission requirements of other non-system applications.

We reviewed the 18 applications' bytecode, searching for any of these three types of error. If we found functionality that could plausibly pertain to a permission that Stowaway identified as unnecessary, we manually wrote additional test cases to confirm the accuracy of our permission map. We investigated the third type of error by checking whether the application sends Intents to pre-installed or well-known applications. When we determined that a warning was not a false positive, we attempted to identify why the developer had added the unnecessary permission.

We also analyzed overprivilege warnings by running the application in our modified version of Android (which records permission checks as they occur) and interacting with it. It was not possible to test all applications at runtime; for example, some applications rely on server-side resources that have moved or changed since we downloaded them. We were able to test 10 of the 18 applications in this way. In each case, runtime testing confirmed the results of our code review.

False Positives. Stowaway identified 18 of the 40 applications (45%) as having 42 unnecessary permissions. Our manual review determined that 17 applications (42.5%) are overprivileged, with a total of 39 unnecessary permissions. This represents a 7% false positive rate.

All three of the false warnings were caused by incompleteness in our permission map. Each was a special case that we failed to anticipate. Two of the three false positives were caused by applications using `Runtime.exec` to execute a permission-protected shell command. (For example, the `logcat` command performs a `READ_LOGS` permission check.) The third false positive was caused by an application that embeds a web site that uses HTML5 geolocation, which requires a location permission. We wrote test cases for these scenarios and updated our permission map.

Of the 40 applications in this set, 4 contain at least one reflective call that our static analysis tool cannot resolve or dismiss. 2 of them are overprivileged. This means that 50% of the applications with at least one unresolved reflective call are overprivileged, whereas other applications are overprivileged at a rate of 42%. However, a sample size of 4 is too small to draw conclusions. We investigated the unresolved reflective calls and do not believe they led to false positives.

Automated Analysis

We ran Stowaway on 900 Android applications. Overall, Stowaway identified 323 applications (35.8%) as having unnecessary permissions. Stowaway was unable to resolve some applications' reflective calls, which might lead to a higher false positive rate in those applications. Consequently, we discuss applications with unresolved reflective calls separately from other applications.

Permission	Usage
ACCESS_NETWORK_STATE	16%
READ_PHONE_STATE	13%
ACCESS_WIFI_STATE	8%
WRITE_EXTERNAL_STORAGE	7%
CALL_PHONE	6%
ACCESS_COARSE_LOCATION	6%
CAMERA	6%
WRITE_SETTINGS	5%
ACCESS_MOCK_LOCATION	5%
GET_TASKS	5%

Table 4.4. The 10 most common unnecessary permissions and the percentage of overprivileged applications that request them.

Applications With Fully Handled Reflection. Stowaway was able to handle all reflective calls for 795 of the 900 applications, meaning that it should have identified all API access for those applications. Stowaway produces overprivilege warnings for 32.7% of the 795 applications. Table 4.4 shows the 10 most common unnecessary permissions among these applications.

56% of overprivileged applications have 1 extra permission, and 94% have 4 or fewer extra permissions. Although one-third of applications are overprivileged, the low degree of per-application overprivilege indicates that developers are attempting to add correct permissions rather than arbitrarily requesting large numbers of unneeded permissions. This supports the potential effectiveness of install-time permission systems like Android's.

We believe that Stowaway should produce approximately the same false positive rate for these applications as it did for the set of 40 that we evaluated in our manual analysis. If we assume that the 7% false positive rate from our manual analysis applies to these results, then 30.4% of the 795 applications are truly overprivileged. Applications could also be *more* overprivileged in practice than indicated by our tool, due to unreachable code. Stowaway does not perform dead code elimination; dead code elimination for Android applications would need to take into account the unique Android lifecycle and application entry points. Additionally, our overapproximation of Content Provider operations (Chapter 4.3.1) might overlook some overprivilege. We did not quantify Stowaway's false negative rate, and we leave dead code elimination and improved Content Provider string tracking to future work.

The Challenges of Java Reflection. Reflection is commonly used in Android applications. Of the 900 applications, 545 (61%) use Java reflection to make API calls. We found that reflection is used for many purposes, such as to deserialize JSON and XML, invoke hidden or private API calls, and handle API classes whose names changed between versions. The prevalence of reflection indicates that it is important for an Android static analysis tool to handle Java reflection, even if the static analysis tool is not intended for obfuscated or malicious code.

Stowaway was able to fully resolve the targets of reflective calls in 59% of the applications that use reflection. We handled a further 117 applications with two techniques: eliminating failures where the target class of the reflective call was known to be defined within the application, and

	Apps with Warnings	Total Apps	Rate
Reflection, failures	56	105	53%
Reflection, no failures	151	440	34%
No reflection	109	355	31%

Table 4.5. The rates at which Stowaway issues overprivilege warnings, by reflection status.

manually examining and handling failures in 21 highly popular libraries. This left us with 105 applications with reflective calls that Stowaway could not resolve or dismiss, which is 12% of the 900 applications.

Stowaway identifies 53.3% of the 105 applications as overprivileged. Table 4.5 compares this to the rate at which warnings are issued for applications without unhandled reflections. There are two possible explanations for the difference: Stowaway might have a higher false positive rate in applications with unresolved reflective calls, or applications that use Java reflection in complicated ways might have a higher rate of actual overprivilege due to a correlated trait.

We suspect that both factors play a role in the higher overprivilege warning rate in applications with unhandled reflective calls. Although our manual review did not find that reflective failures led to false positives, a subsequent review of additional applications identified several erroneous warnings that were caused by reflection. On the other hand, developer error may increase with the complexity associated with complicated reflective calls.

Handling Java reflection is necessary to develop sound and complete program analyses. However, resolving reflective calls is an area of open research. Stowaway's reflection analysis fails when presented with the creation of method names based on non-static environment variables, direct generation of Dalvik bytecode, arrays with two pointers that reference the same location, or `Method` and `Class` objects that are stored in hash tables. Stowaway's primarily linear traversal of a method also experiences problems with non-linear control flow. We also observed several applications that iterate over a set of classes or methods, testing each element to decide which one to invoke reflectively. If multiple comparison values are tested and none are used within the block, Stowaway only tracks the last comparison value beyond the block; this value may be `null`. Future work — particularly, dynamic analysis tools – may be able to solve some of these problems.

Stowaway's results are comparable with past research into resolving reflective calls. Livshits et al. created a static algorithm which approximates reflective targets by tracking string constants passed to reflections [105]. Their approach falls short when the reflective call depends on user input or environment variables. We use the same approach and suffer from the same limitations. They improve their results with developer annotations, which is not a feasible approach for our domain. A more advanced technique combines static analysis with information about the environment of the Java program in order to resolve reflections [130]. However, their results are sound only if the program is executed in an identical environment as the original evaluation. Even with their modifications, they are able to resolve only 74% of reflective calls in the Java 1.4 API. We do not claim to improve the state of the art in resolving Java reflection; instead, we focus on domain-specific heuristics for how reflection is used in Android applications.

Common Developer Errors

In some cases, we are able to determine why developers asked for unnecessary permissions. Here, we consider the prevalence of different types of developer error among the 40 applications from our manual review and the 795 fully handled applications from our automated analysis.

Permission Name. Developers sometimes request permissions with names that sound related to their applications' functionality, even if the permissions are not required. For example, one application from our manual review unnecessarily requests the `MOUNT_UNMOUNT_FILESYSTEMS` permission to receive the `android.intent.action.MEDIA_MOUNTED` Intent. As another example, the `ACCESS_NETWORK_STATE` and `ACCESS_WIFI_STATE` permissions have similar-sounding names, but they are required by different classes. Developers often request them in pairs, even if only one is necessary. Of the applications that unnecessarily request the network permission, 32% legitimately require the WiFi permission. Of the applications that unnecessarily request the WiFi permission, 71% legitimately need the network permission.

Deputies. An application can send an Intent to another *deputy* application, asking the deputy to perform an operation. If the deputy makes a permission-protected API call, then the deputy needs a permission. The sender of the Intent, however, does not. We noticed instances of applications requesting permissions for actions that they asked deputies to do. For example, one application asks the Android Market to install another application. The sender asks for `INSTALL_PACKAGES`, which it does not need because the Market application does the installation.

We find widespread evidence of this type of error. Of the applications that unnecessarily request the `CAMERA` permission, 81% send an Intent that opens the default Camera application to take a picture. 82% of the applications that unnecessarily request `INTERNET` send an Intent that opens a URL in the browser. Similarly, 44% of the applications that unnecessarily request `CALL_PHONE` send an Intent to the default Phone Dialer application.

Related Methods. As shown in Figure 4.3, most classes contain a mix of permission-protected and unprotected methods. We have observed applications that use unprotected methods but request permissions that are required for other methods in the same class. For example, `android.provider.Settings.Secure` is a convenience class in the API for accessing the Settings Content Provider. The class includes both setters and getters. The setters require the `WRITE_SETTINGS` permission, but the getters do not. Two of the applications that we manually reviewed use only the getters but request the `WRITE_SETTINGS` permission.

Copy and Paste. Popular message boards contain Android code snippets and advice about permission requirements. Sometimes this information is inaccurate, and developers who copy it will overprivilege their applications. For example, one of the applications that we manually reviewed registers to receive the `android.net.wifi.STATE_CHANGE` Intent and requests the `ACCESS_WIFI_STATE` permission. As of May 2011, the third-highest Google search result for that Intent contains the incorrect assertion that it requires that permission [2].

Deprecated Permissions. Permissions that are unnecessary in Android 2.2 could be necessary in older Android releases. Old or backwards-compatible applications therefore might have seemingly extra permissions. However, developers may also accidentally use these permissions because they have read out-of-date material. 8% of the overprivileged applications request either `ACCESS_GPS` or `ACCESS_LOCATION`, which were deprecated in 2008. Of those, all but one specify that their lowest supported API version is *higher* than the last version that included those permissions.

Testing Artifacts. A developer might add a permission during testing and then forget to remove it when the test code is removed. For example, `ACCESS_MOCK_LOCATION` is typically used only for testing but can be found in released applications. All of the applications in our data set that unnecessarily include the `ACCESS_MOCK_LOCATION` permission also include a real location permission.

Signature/System Permissions. We find that 9% of overprivileged applications request unneeded Signature or SignatureOrSystem permissions. Standard versions of Android will silently refuse to grant those permissions to applications that are not signed by the device manufacturer. The permissions were either requested in error, or the developers removed the related code after discovering it did not work on standard handsets.

We can attribute many instances of overprivilege to developer confusion over the permission system. Confusion over permission names, related methods, deputies, and deprecated permissions could be addressed with improved API documentation. To avoid overprivilege due to related methods, we recommend listing permission requirements on a per-method (rather than per-class) basis. Confusion over deputies could be reduced by clarifying the relationship between permissions and pre-installed system applications.

Despite the number of unnecessary permissions that we can attribute to error, it is possible that some developers request extra permissions intentionally. Developers are incentivized to ask for unnecessary permissions because, as we have observed, applications will not receive automatic updates if the updated version of the application requests more permissions.

4.4 Overprivilege in Chrome Extensions

We next investigate factors that influence permission requests in Chrome extensions. We establish the rate of overprivilege and then examine the amount of effort that developers are willing to put into using wildcards for domain lists. Last, we consider whether Chrome's extension permission system's low degree of granularity contributes to application overprivilege.

4.4.1 Rate of Overprivilege

As seen in Android (Chapter 4.3.4), developers may ask for unnecessary permissions. We explore the prevalence of overprivilege in Chrome extensions.

Extension Analysis Results

Detecting unnecessary browser managers and domains requires different techniques, so we consider these two forms of overprivilege separately.

Browser Managers. We count the extensions that request browser managers but do not use them. About half of the extensions in our set of 1,000 "popular" extensions request access to security-relevant browser managers. We search their source code (including remotely sourced scripts) for references to their requested browser managers. 14.7% of the 1,000 extensions are overprivileged by this measure because they request access to managers that they never use. It is possible for an extension to name a browser manager without explicitly including the name as a string (e.g., `"book"+"marks"`); we examined a random sample of 15 overprivileged extensions and found no evidence of developers doing this.

Domains. We also review fifty randomly selected extensions for excessive domain access (see Appendix A.1). For each extension, we compare the permissions it requests with the domains needed to implement its functionality, which we determine by manually exercising the user interface and consulting its source code when necessary. We find that 41 of the 50 extensions request access to web data, and 7 of those are overprivileged: 5 request too many domain permissions for their core extensions, and 2 install content scripts on unnecessary domains.

The reasons for overprivilege are diverse. One example is "PBTweet+", which requests web access for a nonexistent core extension; other examples are "iBood" and "Castle Age Autoplayer", which request access to all domains even though they only interact with `ibood.com` and `facebook.com`, respectively. "Send using Gmail (no button)" demonstrates a common error, which is that developers sometimes request access to all and specific domains in the same list. The author of the "Send using Gmail (no button)" extension wrote the following specific domain list: `"http://*/*", "https://*/*", "http://*.google.com/"`. We find that an additional 27 of the 1,000 popularity-ranked extensions also make this mistake. This is a conservative measure of wildcard-induced error; subdomain wildcards can feature the same mistake, like asking for both `http://www.example.com` and `http://*.example.com`.

Tools for Error Reduction

Our Google Chrome extension overprivilege detection tool is simple but sufficient to find some types of errors. A JavaScript text search (i.e., a set of regular expressions) is sufficient to remove unnecessary browser manager permissions from 147 of the 1,000 popularity-ranked extensions. Our text search can potentially have false positives; when manually verifying the results of our tool, we identified three extensions that only contain references to browser managers in remotely sourced scripts. However, a developer can disregard a warning if she feels it is incorrect. Our tool also detects simple redundant wildcard errors and asks the developer to remove the broad wildcard in favor of the more specific domain. Detecting the larger problem of overly broad domain requests is a challenging open problem for future research in JavaScript program analysis.

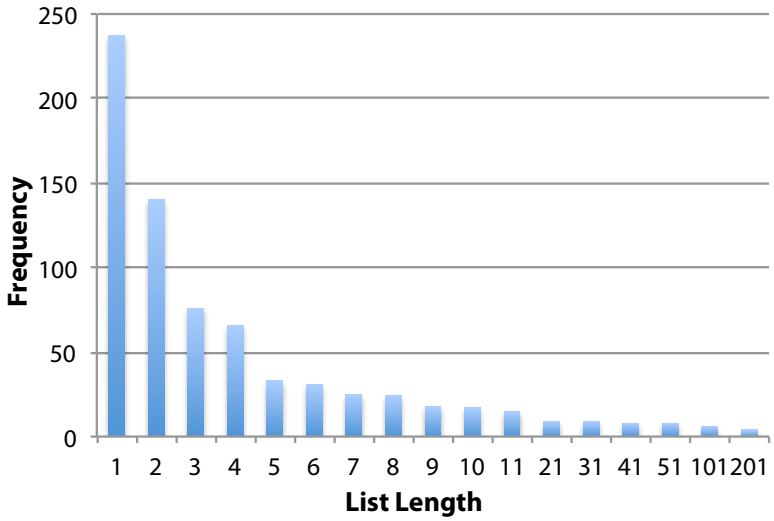


Figure 4.4. The number of content script specific domain lists with *at least* a given length.

4.4.2 Developer Effort

Domain access in the Google Chrome extension system relies on wildcards. A developer can write `<all_urls>` or `*://*/*` and gain access to all domains, or she can define a list of specific domains. When it is not feasible for a developer to list all possible subdomains, she can use wildcards to capture multiple subdomains. However, a developer might choose to use a wildcard even though it includes more privileges than the application requires.

Compliance. To determine whether developers are willing to write specific domain lists when they can more easily request access to all domains, we evaluate the prevalence of specific domain lists in the 1,000 popularity-ranked extensions. Of the 714 extensions that need access to web data, 428 use a specific domain list for either a content script or core extension. This is a surprising and positive finding: 60% of developers whose extensions need web access choose to opt in to domain restrictions for at least one component. However, 367 extensions also have at least one component that requests full domain access. (An extension with multiple content scripts might request full domain access for some scripts but place restrictions on others.)

Amount of Effort. We suspect that developers will default to requesting all-domain access if the number of specific domains in the list grows too high. To examine this further, we consider the 237 content scripts that use specific domain lists. The lists are short: only 31 are longer than five. Figure 4.4 presents the distribution. This indicates that most developers either request a very small number of domains or opt to request full domain access, with few in-between. However, six developers wrote eight lists that are longer than fifty domains. These outliers result from developers internationalizing their extensions by repeating the same domains with different suffixes; wildcards cannot be used to represent suffixes because the domains may have different owners.

Noncompliance. Chapter 4.4.1 describes a manual analysis of fifty extensions. Five of those extensions are overprivileged due to improper wildcard use. Two of the developers choose to request all-domain access rather than write specific domain lists, two write specific domain lists but unnecessarily use wildcards for subdomains, and one incorrectly requests all-domain access alongside specific domains. In other words, 12% of the extensions with web access request excessive permissions because their developers are unable or unwilling to write specific domain lists.

In summary, our findings are twofold. We show that 60% of extension developers write at least one specific domain list. This demonstrates that the option to write specific domain lists is a worthwhile part of a declarative permission system. On the other hand, 40% of developers whose extensions need web access do not write any specific domain lists. Furthermore, our manual analysis indicates that 12% of extensions with web access use wildcards improperly.

4.4.3 Permission Granularity

If a single permission protects a diverse set of API calls, then an application seeking to use only a subset of that functionality will be overprivileged even though the developer requested the correct permissions. Separating a coarse permission into multiple permissions can improve the correlation between permissions and application functionality. On the other hand, excessively fine-grained permissions would burden developers.

At the time of our study, Google Chrome extension permissions were at the granularity of a browser manager: one permission per entire browser manager. This posed a problem for the window manager, which includes some methods that provide indirect access to history via the location property of loaded windows. (In other words, an extension with the window manager permission cannot access the history manager, but it can observe the address of every page as it is loaded.) Using the window manager generated history warnings, regardless of whether the extension used any of the methods that provide access to the location property. The fact that the window manager caused a history warning was confusing to users and developers. Consider this quote from the developer of *Neat Bookmarks*:

> Installing this extension will ask for permission to access your browsing history, which is totally useless, not used and not stored by the extension at all. Not really sure why 'History' is part of 'Bookmarks' in the Chrome browser.

The developer is so confused by the history warning that he or she believes it is caused by the extension's use of the bookmark manager, rather than the window manager.

Since the time of our study, the window manager has been changed so that certain methods do not require any permission [11]. Consequently, developers can access some non-history-related functionality without acquiring a permission that shows users the history warning.

4.5 Other Overprivilege Considerations

Not all instances of overprivilege are caused by developer error. In some cases, developer incentives can encourage or discourage the rate of overprivilege.

Review Process. Formal review can delay an application's entrance into the directory. Developers are often concerned about the length of the review process [12]. If dangerous permissions increase the review time (and a lack of dangerous permissions decreases it), then developers have an incentive to use as few permissions as necessary. Google Chrome extensions have to undergo a review process if they include plug-ins, which incentivizes developers to not use plug-ins. Other platforms could adopt similar systems or institute a timed delay for applications with more permissions.

Pressure From Users. The ultimate goal of a developer is to reach as many users as possible. If users are hesitant to install applications with certain permissions, then developers are motivated to avoid those permissions. Users can express their dislike of permission requests in application comments and e-mails to the developer.

To illustrate user pressure, we read the user comments for 50 randomly selected Google Chrome extensions with at least one permission. Of the 50 extensions, 8 (15%) have at least one comment questioning the extension's use of permissions. A majority of the critical permission comments refer to the extension's ability to access browsing history. Several commenters state that the permission requests are preventing them from installing an application, e.g., "Really would like to give it a try. ... But why does it need access to my history? I hope you got a plausible answer because I really would like to try it out." These comments indicate that a small number of vocal users are pressuring developers to use fewer permissions. Additionally, the developers of 3 extensions include an explanation of their permission usage in the extension description. This indicates that these developers are concerned about user reactions to permission requests.

Automatic Updates. Android and Google Chrome will automatically update applications as they become available, according to user preferences. However, automatic updates will not proceed for applications whose updates request more permissions. Instead, the user needs to manually install the update and approve the new permissions; in Android, this amounts to several additional screens. This incentivizes developers to request unnecessary permissions in the first version, in case later versions require the permissions. The user interface for updates could be improved to minimize the user effort required to update applications with new permissions.

4.6 Conclusion

This chapter contributes evidence in support of application permission systems. Our large-scale analysis of Google Chrome extensions and Android applications finds that applications ask for significantly fewer than the maximum set of permissions. Only 14 of 1,000 Google Chrome extensions use native code, which is the most dangerous privilege. Approximately 30% of extension

developers restrict their extensions' web access to a small set of domains. All Android applications ask for less than half of the available Dangerous permissions, and a majority request less than 4.

We developed a tool, Stowaway, that detects overprivilege in Android applications. We applied Stowaway to 940 Android applications and found that about one-third are overprivileged. Applications generally are overprivileged by only a few permissions, and overprivilege can often be attributed to developer confusion. We also reviewed Chrome extensions and found that many developers are willing to take the time to create specific domain lists. These studies demonstrate that most developers are willing to attempt to follow least privilege.

These findings indicate that permission systems with up-front permission declarations have an advantage over the traditional user permission model because the impact of a potential third-party vulnerability is greatly reduced when compared to a full-privilege system. These results can be extended to platforms with runtime consent dialogs if the platform requires developers to declare a set of maximum permissions. Chapter 5 further explores the potential benefit of defense-in-depth by examining vulnerable extensions.

However, our study shows that users are frequently presented with requests for dangerous permissions during application installation in install-time systems. As a consequence, installation security warnings may not be an effective malware prevention tool, even for alert users. Chapter 6 further evaluates the usability of Android permissions.

4.7 Acknowledgements

We thank our collaborators for contributing to this work. Kate Greenwood helped review extensions' wildcard lists (Chapter 4.4.2). Erika Chin worked on testing the Android API and building Stowaway; in particular, she implemented the inter-procedural static analysis to handle Java reflection (Chapter 4.3). Steve Hanna, Royce Cheng-Yue, and Kathryn Lingel also tested Content Providers as part of the Android overprivilege analysis (Chapter 4.3).

Chapter 5

The Effect of Permissions on Chrome Extension Vulnerabilities

5.1 Introduction

Vulnerabilities in browser extensions put users at risk by providing a way for website and network attackers to gain access to users' private data and credentials. Developers need to build extensions that are robust to attacks originating from malicious websites and the network. In 2009, Google Chrome introduced a new extension platform with several features intended to prevent and mitigate extension vulnerabilities:

- *Permissions*. Each extension comes packaged with a list of permissions, which govern access to the browser APIs and web domains. If an extension has a core extension vulnerability, the attacker will only gain access to the permissions that the vulnerable extension already has.
- *Privilege separation*. Chrome extensions adhere to a privilege-separated architecture [122]. Extensions are built from two types of components, which are isolated from each other: *content scripts* and *core extensions*. Content scripts interact with websites and execute with no privileges. Core extensions do not directly interact with websites and execute with the extension's full privileges.
- *Isolated worlds*. Content scripts can read and modify website content, but content scripts and websites have separate program heaps so that websites cannot access content scripts' functions or variables.

In this chapter, we provide an empirical analysis of these security mechanisms, which together comprise a state-of-the-art least privilege system. We analyze 100 Chrome extensions, including the 50 most popular extensions, to determine whether Chrome's security mechanisms successfully prevent or mitigate extension vulnerabilities. We find that 40 extensions contain at least one type of vulnerability. Twenty-seven extensions contain core extension vulnerabilities, which give an attacker full control over the extension.

Based on this set of vulnerabilities, we evaluate the effectiveness of each of the three security mechanisms. Our primary findings are:

- Permissions significantly reduce the severity of half of the core extension vulnerabilities, which demonstrates that permissions are effective at mitigating vulnerabilities in practice. Additionally, dangerous permissions do not correlate with vulnerabilities: developers who write vulnerable extensions use permissions the same way as other developers.
- The isolated worlds mechanism is successful at preventing content script vulnerabilities.
- The success of the isolated worlds mechanism renders privilege separation unnecessary. However, privilege separation would protect 62% of extensions if isolated worlds were to fail. In the remaining 38% of extensions, developers either intentionally or accidentally negate the benefits of privilege separation. This highlights that forcing developers to divide their software into components does not automatically achieve security on its own.

Although these mechanisms reduce the rate and scope of several classes of attacks, a large number of high-privilege vulnerabilities remain.

We propose and evaluate four additional defenses. Our extension review demonstrates that many developers do not follow security best practices if they are optional, so we propose four mandatory bans on unsafe coding practices. We quantify the security benefits and functionality costs of these restrictions on extension behavior. Our evaluation shows that banning inline scripts and HTTP scripts would prevent 67% of the overall vulnerabilities and 94% of the most dangerous vulnerabilities at a relatively low cost for most extensions. In concurrent work, Google Chrome implemented Content Security Policy (CSP) for extensions to optionally restrict their own behavior. Motivated in part by our study, new versions of Chrome use CSP to enforce some of the mandatory bans that we proposed and evaluated [32].

5.2 Threat Model

In this chapter, we focus on non-malicious extensions that are vulnerable to external attacks. Most extensions are written by well-meaning developers who are not security experts. We do not consider malicious extensions; preventing malicious extensions requires completely different tactics, such as warnings, user education, security scans of the market, and feedback and rating systems. Benign-but-buggy extensions face two types of attacks:

- *Network attackers.* People who use insecure networks (e.g., public WiFi hotspots) may encounter network attackers [129, 110]. A network attacker's goal is to obtain personal information or credentials from a target user. To achieve this goal, a network attacker will read and alter HTTP traffic to mount man-in-the-middle attacks. (Assuming that TLS works as intended, a network attacker cannot compromise HTTPS traffic.) Consequently, data and scripts loaded over HTTP may be compromised.

  If an extension adds an *HTTP script* — a JavaScript file loaded over HTTP — to itself, a network attacker can run arbitrary JavaScript within the extension's context. If an extension adds an HTTP script to an HTTPS website, then the website will no longer benefit from the confidentiality, integrity, and authentication guarantees of HTTPS. Similarly, inserting HTTP data into an HTTPS website or extension can lead to vulnerabilities if the untrusted data is allowed to execute as code.

- *Web attackers.* Users may visit websites that host malicious content (e.g., advertisements or user comments). A website can launch a cross-site scripting attack on an extension if the extension treats the website's data or functions as trusted. The goal of a web attacker is to gain access to users' data in the browser (e.g., history) or violate website isolation (e.g., read another site's password).

Extensions are primarily written in JavaScript and HTML, and JavaScript provides several methods for converting strings to code, such as `eval` and `setTimeout`. If used improperly, these methods can introduce code injection vulnerabilities that compromise the extension. Data can also execute if it is written to a page as HTML instead of as text, e.g., through the use of `document.write` or `document.body.innerHTML`. Extension developers need to be careful to avoid passing unsanitized, untrusted data to these execution sinks.

5.3 Extension Security Review

We reviewed 100 Google Chrome extensions from the official directory. This set is comprised of the 50 most popular extensions and 50 randomly-selected extensions from June 2011.[1] Chapter 5.3.1 presents our extension review methodology. Our security review found that 40% of the extensions contain vulnerabilities, and Chapter 5.3.2 describes the vulnerabilities. Chapter 5.3.3 presents our observation that 31% of developers do not follow even the simplest security best practices. We notified most of the authors of vulnerable extensions (Chapter 5.3.4).

[1] We excluded four extensions because they included binary plugins, which require different analysis techniques; they were replaced with the next popular or random extensions. The directory's popularity metric is primarily based on the number of users.

5.3.1 Methodology

We manually reviewed the 100 selected extensions, using a three-step security review process:

1. *Black-box testing.* We exercised each extension's user interface and monitored its network traffic to observe inputs and behavior. We looked for instances of network data being inserted into the DOM of a page. After observing an extension, we inserted malicious data into its network traffic (including the websites it interacts with) to test potential vulnerabilities.

2. *Source code analysis.* We examined extensions' source code to determine whether data from an untrusted source could flow to an execution sink. After manually reviewing the source code, we used `grep` to search for any additional sources or sinks that we might have missed. For sources, we looked for static and dynamic script insertion, `XMLHttpRequest`s, cookies, bookmarks, and reading websites' DOMs. For sinks, we looked for uses of `eval`, `setTimeout`, `document.write`, `innerHTML`, etc. We then manually traced the call graph to find additional vulnerabilities.

3. *Holistic testing.* We matched extensions' source code to behaviors we identified during black-box testing. With our combined knowledge of an extension's source code, network traffic, and user interface, we attempted to identify any additional behavior that we had previously missed.

We then verified that all of the vulnerabilities could occur in practice by building attacks. Our goal was to find all vulnerabilities in every extension.

During our review, we looked for three types of vulnerabilities: vulnerabilities that extensions add to websites (e.g., HTTP scripts on HTTPS websites), vulnerabilities in content scripts, and vulnerabilities in core extensions. Some content script vulnerabilities may also be core extension vulnerabilities, depending on the extensions' architectures. Core extension vulnerabilities are the most severe because the core is the most privileged extension component. We do not report vulnerabilities if the potential attacker is a trusted website (e.g., `https://mail.google.com`) and the potentially malicious data is not user-generated; we do not believe that well-known websites are likely to launch web attacks.

After our manual review, we applied a commercial static analysis tool to six extensions, with custom rules. However, our manual review identified significantly more vulnerabilities, and the static analysis tool did not find any additional vulnerabilities because of limitations in its ability to track strings through manipulations. Prior research has similarly found that a manual review by experts uncovers more bugs than static analysis tools [149]. Our other alternative, VEX [27], was not built to handle several of the types of attacks that we included in our threat model. Consequently, we did not pursue static analysis further.

Vulnerable Component	Web Attacker	Network Attacker
Core extension	5	50
Content script	3	1
Website	6	14

Table 5.1. 70 vulnerabilities, by location and threat model.

Vulnerable Component	Popular	Random	Total
Core extension	12	15	27
Content script	1	2	3
Website	11	6	17
Any	22	18	40

Table 5.2. The number of extensions with vulnerabilities, of 50 popular and 50 randomly-selected extensions.

5.3.2 Vulnerabilities

We found 70 vulnerabilities across 40 extensions. Appendix A.2 identifies the vulnerable extensions. Table 5.1 categorizes the vulnerabilities by the location of the vulnerability and the type of attacker that could exploit it. More of the vulnerabilities can be leveraged by a network attacker than by a web attacker, which reflects the fact that two of the Chrome extension platform's security measures were primarily designed to prevent web attacks. A bug may be vulnerable to both web and network attacks; we count it as a single vulnerability but list it in both categories in Table 5.1 for illustrative purposes.

The vulnerabilities are evenly distributed between popular and randomly-selected extensions. Table 5.2 shows the distribution. Although popular extensions are more likely to be professionally written, this does not result in a lower vulnerability rate in the set of popular extensions that we examined. We hypothesize that popular extensions have more complex communication with websites and servers, which increases their attack surface and neutralizes the security benefits of having been professionally developed. The most popular extension with a vulnerability had 768,154 users in June 2011.

5.3.3 Developer Security Effort

Most extension developers are not security experts. However, there are two best practices that a security-conscious extension developer can follow without any expertise. First, developers can use HTTPS instead of HTTP when it is available, to prevent a network attacker from inserting data or code into an extension. Second, developers can use `innerText` instead of `innerHTML` when adding untrusted, non-HTML data to a page; `innerText` does not allow inline scripts to execute.

We evaluate developers' use of these best practices to determine whether they are security-conscious. We find that 31 extensions contain at least one vulnerability that was caused by not following these two simple best practices. This demonstrates that a substantial fraction of developers do not make use of optional security mechanisms, even if the security mechanisms are very simple to understand and use. As such, we advocate mandatory security mechanisms that force developers to follow best security practices (Chapter 5.7).

5.3.4 Author Notification

We disclosed the extensions' vulnerabilities to all of the developers that we were able to contact. We found contact information for 80% of the vulnerable extensions.[2] Developers were contacted between June and September 2011, depending on when we completed each review. We sent developers follow-up e-mails if they did not respond to our initial disclosure within a month.

Of the 32 developers that we contacted, 19 acknowledged and fixed the vulnerabilities in their extensions, and 7 acknowledged the vulnerabilities but have not fixed them as of February 7, 2012. Two of the un-patched extensions are official Google extensions. As requested, we provided guidance on how the security bugs could be fixed. None of the developers disputed the legitimacy of the vulnerabilities, although one developer argued that a vulnerability was too difficult to fix because it would require moving some content to child frames.

Appendix A.2 identifies the extensions that have been fixed. However, the "fixed" extensions are not necessarily secure despite our review. While checking on the status of vulnerabilities, we discovered that developers of several extensions have introduced new security vulnerabilities that were not present during our initial review. We do not discuss the new vulnerabilities in this chapter.

5.4 Evaluation of the Permission System

The Chrome permission system is intended to reduce the severity of core extension vulnerabilities. If a website or network attacker were to successfully inject malicious code into a core extension, the severity of the attack would be limited by the extension's permissions. However, permissions will not mitigate vulnerabilities in extensions that request many dangerous permissions. We evaluate the extent to which permissions mitigate the core extension vulnerabilities that we found.

Table 5.3 lists the permissions that the vulnerable extensions request. Ideally, each permission should be requested infrequently. We find that 70% of vulnerable extensions request the `tabs` permission; an attacker with access to the `tabs` API can collect a user's browsing history or redirect pages that a user views. Fewer than half of extensions request each of the other permissions.

[2]For the remaining 20%, contact information was unavailable, the extension had been removed from the directory, or we were unable to contact the developer in a language spoken by the developer.

Permissions	Times Requested	Percentage
tabs (browsing history)	19	70%
all HTTP domains	12	44%
all HTTPS domains	12	44%
specific domains	10	37%
notifications	5	19%
bookmarks	4	15%
no permissions	4	15%
cookies	3	11%
geolocation	1	4%
context menus	1	4%
unlimited storage	1	4%

Table 5.3. The permissions that are requested by the 27 extensions with core extension vulnerabilities.

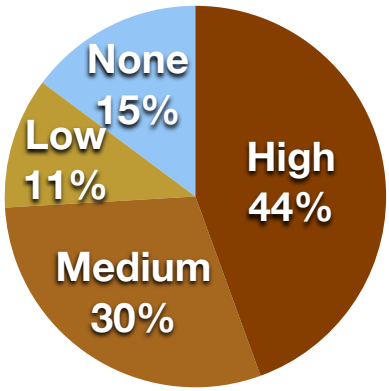


Figure 5.1. The 27 extensions with core vulnerabilities, categorized by the severity of their worst vulnerabilities.

To summarize the impact of permissions on extension vulnerabilities, we categorized all of the vulnerabilities by attack severity. We based our categorization on the Firefox Security Severity Ratings [19], which has been previously used to classify extension privileges [31]:

- *Critical*: Leaks the permission to run arbitrary code on the user's system
- *High*: Leaks permissions for the DOM of all HTTP(S) websites
- *Medium*: Leaks permissions for private user data (e.g., history) or the DOM of specific websites that contain financial or important personal data (e.g., `https://*.google.com/*`)
- *Low*: Leaks permissions for the DOM of specific websites that do not contain sensitive data (e.g., `http://*.espncricinfo.com`) or permissions that can be used to annoy the user (e.g., fill up storage or make notifications)

- *None*: Does not leak any permissions

We did not find any critically-vulnerable extensions. This is a consequence of our extension selection methodology: we did not review any extensions with binary plugins, which are needed to obtain critical privileges.

Figure 5.1 categorizes the 27 vulnerable extensions by their most severe vulnerabilities. In the absence of a permission system, all of the vulnerabilities would give an attacker access to all of the browser's privileges (i.e., critical privileges). With the permission system, less than half of the vulnerable extensions yield access to high-severity permissions. As such, our study demonstrates that the permission system successfully limits the severity of most vulnerabilities.

We hypothesized that permissions would positively correlate with vulnerabilities. Chapter 4 showed that many extensions are over-permissioned, and we thought that developers who are unwilling to follow security best practices (e.g., use HTTPS) would be unwilling to take the time to specify the correct set of permissions. This would result in vulnerable extensions requesting dangerous permissions at a higher rate. However, we do not find any evidence of a positive correlation between vulnerabilities and permissions. The 27 extensions with core vulnerabilities requested permissions at a lower rate than the other 73 extensions, although the difference was not statistically significant. Our results show that developers of vulnerable extensions can use permissions well enough to reduce the privileges of their insecure extensions, even though they lack the expertise or motivation required to secure their extensions.

Permissions are not only used by the Google Chrome extension system. Android implements a similar permission system, and future HTML5 device APIs will likely be guarded with permissions. Although it has been assumed that permissions mitigate vulnerabilities (Chapters 4 and [76]), our study is the first to evaluate whether this is true for real-world vulnerabilities or measure quantitatively how much it helps mitigate these vulnerabilities in practice. Our findings indicate that permissions can have a significant positive impact on the system's security and are worth including in a new platform as a second line of defense against attacks. However, they are not effective enough to be relied on as the only defense mechanism.

5.5 Evaluation of Isolated Worlds

The isolated worlds mechanism is intended to protect content scripts from malicious websites, including otherwise-benign websites that have been altered by a network attacker. We evaluate whether the isolated worlds mechanism is sufficient to protect content scripts from websites. Our security review indicates that isolated worlds largely succeeds: only 3 of the 100 extensions have content script vulnerabilities, and only 2 of the vulnerabilities allow arbitrary code execution.

Developers face four main security challenges when writing extensions that interact with websites. We discuss whether and how well the isolated worlds mechanism helps prevent these vulnerability classes.

Data as HTML. One potential web development mistake is to insert untrusted data as HTML into a page, thereby allowing untrusted data to run as code. The isolated worlds mechanism mitigates this type of error in content scripts. When a content script inserts data as HTML into a website, any scripts in the data are executed within the website's isolated world instead of the extension's. This means that an extension can read data from a website's DOM, edit it, and then re-insert it into the page without introducing a content script vulnerability. Alternately, an extension can copy data from one website into another website. In this case, the extension will have introduced a vulnerability into the edited website, but the content script itself will be unaffected.

We expect that content scripts would exhibit a higher vulnerability rate if the isolated worlds mechanism did not mitigate data-as-HTML bugs. Six extensions' content scripts contained data-as-HTML errors that resulted in web site vulnerabilities, instead of the more-dangerous content script vulnerabilities. Content scripts have strictly more privileges than web sites, so content script vulnerabilities are more severe; content scripts have the additional ability of communicating with the core extension. Consequently, we conclude that the isolated worlds mechanism reduces the rate of content script vulnerabilities by reducing data-as-HTML errors to web site vulnerabilities.

Eval. Developers can introduce vulnerabilities into their extensions by using `eval` to execute untrusted data. If an extension reads data from a website's DOM and `eval`s the data in a content script, the resulting code will run in the content script's isolated world. As such, the isolated worlds mechanism does not prevent or mitigate vulnerabilities due to the use of `eval` in a content script.

We find that relatively few developers use `eval`, possibly because its use has been responsible for well-known security problems in the past [46, 13]. Only 14 extensions use `eval` or equivalent constructs to convert strings to code in their content scripts, and most of those use it only once in a third-party library function. However, we did find two content script vulnerabilities that arise because of an extension's use of `eval` in its content script. For example, the *Blank Canvas Script Handler* extension can be customized with supplemental scripts, which the extension downloads from a website and `eval`s in a content script. Although the developer is intentionally running data from the website as code, the integrity of the HTTP website that hosts the supplemental scripts could be compromised by a network attacker.

Click Injection. Extensions can register event handlers for DOM elements on websites. For example, an extension might register a handler for a button's `onClick` event. However, extensions cannot differentiate between events that are triggered by the user and events that are generated by a malicious web site. A website can launch a click injection attack by invoking an extension's event handler, thereby tricking the extension into performing an action that was not requested by the user. Although this attack does not allow the attacker to run arbitrary code in the vulnerable content script, it does allow the website to control the content script's behavior.

The isolated worlds mechanism does not prevent or mitigate click injection attacks at all. However, the attack surface is small because relatively few extensions register event handlers for websites' DOM elements. Of the 17 extensions that register event handlers, most are for simple buttons that toggle UI state. We observed only one click injection vulnerability, in the *Google Voice* extension. The extension changes phone numbers on websites into links. When a user clicks a phone number link, Google Voice inserts a confirmation dialog onto the DOM of the website to ensure that the user wants to place a phone call. Google Voice will place the call following the user's con-

firmation. However, a malicious website could fire the extension's event handlers on the link and confirmation dialog, thereby placing a phone call from the user's Google Voice account without user consent.

Prototypes and Capabilities. In the past, many vulnerabilities due to prototype poisoning and capability leaks have been observed in bookmarklets and Firefox extensions [104, 154, 20]. *Prototype poisoning* occurs when a website maliciously alters the behavior of native objects by changing their prototypes [20]. The isolated worlds mechanism provides heap separation, which prevents both of these types of attacks. Regardless of developer behavior, these attacks are not possible in Chrome extensions as long as the isolation mechanism works correctly.

Based on our security review, the isolated worlds mechanism is highly effective at shielding content scripts from malicious websites. It mitigates data-as-HTML errors, which we found were very common in the Chrome extensions that we reviewed. Heap separation also prevents prototype poisoning and capability leaks, which are common errors in bookmarklets and Firefox extensions. Although the isolated worlds mechanism does not prevent click injection or `eval`-based attacks, we find that developers rarely make these mistakes. We acknowledge that our manual review could have missed some content script vulnerabilities. However, we find it unlikely that we could have missed many, given our success at finding the same types of vulnerabilities in core extensions. We therefore conclude that the isolated worlds mechanism is effective, and other extension platforms should implement it if they have not yet done so.

5.6 Evaluation of Privilege Separation

Privilege separation is intended to shield the privileged core extension from attacks. The isolated worlds mechanism serves as the first line of defense against malicious websites, and privilege separation is supposed to protect the core extension when isolated worlds fails. We evaluate the effectiveness of extension privilege separation and find that, although it is unneeded, it would be partially successful at accomplishing its purpose if the isolated worlds mechanism were to fail.

5.6.1 Cross-Component Vulnerabilities

Some developers give content scripts access to core extension permissions, which removes the defense-in-depth benefits of privilege separation. We evaluate the impact of developer behavior on the effectiveness of extension privilege separation.

Vulnerable Content Scripts. The purpose of privilege separation is to limit the impact of content script vulnerabilities. Even if a content script is vulnerable, privilege separation should prevent an attacker from executing code with the extension's permissions. We identified two extensions with content script vulnerabilities that permit arbitrary code execution; these two extensions could benefit from privilege separation.

Permissions	Number of Scripts
All of the extension's permissions	4
Partial: Cross-origin XHRs	9
Partial: Tab control	5
Partial: Other	5

Table 5.4. 61 extensions have content scripts that do not have code injection vulnerabilities. If an attacker were hypothetically able to compromise the content scripts, these are the permissions that the attacker could gain access to via the message-passing channel.

Despite privilege separation, both of the vulnerabilities yield access to some core extension privileges. The vulnerable content scripts can send messages to their respective core extensions, requesting that the core extensions exercise their privileges. In both extensions, the core extension makes arbitrary XHRs on behalf of the content script and returns the result to the content script. This means that the two vulnerable content scripts could trigger arbitrary HTTP XHRs even though content scripts should not have access to a cross-origin `XMLHttpRequest` object. These vulnerable extensions represent a partial success for privilege separation because the attacker cannot gain full privileges, but also a partial failure because the attacker can gain the ability to make cross-origin XHRs.

Hypothetical Vulnerabilities. Due to the success of the isolated worlds mechanism, our set of vulnerabilities only includes two extensions that need privilege separation as a second line of defense. To expand the scope of our evaluation of privilege separation, we explore a hypothetical scenario: if the currently-secure extensions' content scripts had vulnerabilities, would privilege separation mitigate these vulnerabilities?

Of the 98 extensions that do not have content script vulnerabilities, 61 have content scripts. We reviewed the message passing boundary between these content scripts and their core extensions. We determined that 38% of content scripts can leverage communication with their core extensions to abuse some core extension privileges: 4 extensions' content scripts can use all of their cores' permissions, and 19 can use some of their cores' permissions. Table 5.4 shows which permissions attackers would be able to obtain via messages if they were able to compromise the content scripts.[3] This demonstrates that privilege separation could be a relatively effective layer of defense, if needed: we can expect that privilege separation would be effective at limiting the damage of a content script vulnerability 62% of the time.

Example. The *AdBlock* extension allows its content script to execute a set of pre-defined functions in the core extension. To do this, the content script sends a message to the core extension. A string in the message is used to index the `window` object, allowing the content script to select a pre-defined function to run. Unfortunately, this also permits arbitrary code execution because the `window` object provides access to `eval`. As such, a compromised content script would have unfettered access to the core extension's permissions.

[3]In newer versions of Chrome, content scripts can directly make cross-origin XHRs. However, this was not permitted at the time of our study, so we consider these cases as potential privilege escalation vulnerabilities.

Example. A bug in the *Web Developer* extension unintentionally grants its content script full privileges. Its content script can post small notices to the popup page, which is part of the core extension. The notices are inserted using `innerHTML`. The notices are supposed to be text, but a compromised content script could send a notice with an inline script that would execute in the popup page with full core extension permissions.

5.6.2 Web Site Metadata Vulnerabilities

The Chrome extension platform applies privilege separation with the expectation that malicious website data will first enter an extension via a vulnerable content script. However, it is possible for a website to attack a core extension without crossing the privilege separation boundary. Website-controlled metadata such as titles and URLs can be accessed by the core extension through browser managers (e.g., the history, bookmark, and tab managers). This metadata may include inline scripts, and mishandled metadata can lead to a core extension vulnerability. Website metadata does not flow through content scripts, so privilege separation does not impede it. We identified five vulnerabilities from metadata that would allow an attacker to circumvent privilege separation.

Example. The *Speeddial* extension replicates Chrome's built-in list of recently closed pages. Speeddial keeps track of the tabs opened using the tabs manager and does not sanitize the titles of these pages before adding them to the HTML of one of its core extension pages. If a title were to contain an inline script, it would execute with the core extension's permissions.

5.6.3 Direct Network Attacks

Privilege separation is intended to protect the core extension from web attackers and HTTP websites that have been compromised by network attackers. However, the core extension may also be subject to direct network attacks. Nothing separates a core extension from code in HTTP scripts or data in HTTP `XMLHttpRequest`s. HTTP scripts in the core extension give a network attacker the ability to execute code with the extension's full permissions, and HTTP XHRs cause vulnerabilities when extensions allow the HTTP data to execute.

Direct network attacks comprise the largest class of core extension vulnerabilities, as Table 5.5 illustrates. Of the 50 core extension vulnerabilities, 44 vulnerabilities (88%) stem from HTTP scripts or HTTP `XMLHttpRequests`, as opposed to website data. For example, many extensions put the HTTP version of the Google Analytics script in the core extension to track which of the extensions' features are used.

Example. Google Dictionary allows a user to look up definitions of words by double clicking on a word. The desired definition is fetched by making a HTTP request to `google.com` servers. The response is inserted into one of the core extension's pages using `innerHTML`. A network attacker could modify the response to contain malicious inline scripts, which would then execute as part of the privileged core extension page.

Type	Vulnerabilities
Website content	2
Website metadata	5
HTTP XHR	16
HTTP script	28
Total	50

Table 5.5. The types of core extension vulnerabilities.

5.6.4 Implications

The isolated worlds mechanism is so effective at protecting content scripts from websites that privilege separation is rarely needed. As such, privilege separation is used to address a threat that almost does not exist, at the cost of increasing the complexity and performance overhead of extensions. (Privilege separation requires an extra process for each extension, and communication between content scripts and core extensions is IPC.) We find that network attackers are the real threat to core extension security, but privilege separation does not mitigate or prevent these attacks. This shows that although privilege separation can be a powerful security mechanism [122], its placement within an overall system is an important determining factor of its usefulness.

Our study also has implications for the use of privilege separation in other contexts. All Chrome extension developers are required to privilege separate their extensions, which allows us to evaluate how well developers who are not security experts use privilege separation. We find that privilege separation would be fairly effective at preventing web attacks in the absence of isolated worlds: privilege separation would fully protect 62% of core extensions. However, in more than a third of extensions, developers created message passing channels that allow low-privilege code to exploit high-privilege code. This demonstrates that forcing developers to privilege separate their software will improve security in most cases, but a significant fraction of developers will accidentally or intentionally negate the benefits of privilege separation. Mandatory privilege separation could be a valuable line of defense for another platform, but it should not be relied on as the only security mechanism; it should be coupled with other lines of defense.

5.7 Defenses

Despite Google Chrome's security architecture, our security review identified 70 vulnerabilities in 40 extensions. Based on the nature of these vulnerabilities, we propose and evaluate four additional defenses. The defenses are bans on unsafe coding practices that lead to vulnerabilities. We advocate mandatory bans on unsafe coding practices because many developers do not follow security best practices when they are optional (Chapter 5.3.3). We quantify the security benefits and compatibility costs of each of these defenses to determine whether they should be adopted. Our main finding is that a combination of banning HTTP scripts and banning inline scripts would

Restriction	Security Benefit	Broken, But Fixable	Broken And Unfixable
No HTTP scripts in core	15%	15%	0%
No HTTP scripts on HTTPS websites	8%	8%	0%
No inline scripts	15%	79%	0%
No `eval`	3%	30%	2%
No HTTP XHRs	17%	29%	14%
All of the above	35%	86%	16%
No HTTP scripts and no inline scripts	32%	80%	0%
Chrome 18 policy	27%	85%	2%

Table 5.6. The percentage of the 100 extensions that would be affected by the restrictions. The "Security Benefit" column shows the number of extensions that would be fixed by the corresponding restriction.

prevent 94% of the core extension vulnerabilities, with only a small amount of developer effort to maintain full functionality in most cases.

In concurrent work, Google Chrome implemented Content Security Policy (CSP) for extensions. CSP can be used to enforce all four of these defenses. Initially, the use of CSP was wholly optional for developers. As of Chrome 18, extensions that take advantage of new features will be subject to a mandatory policy; this change was partially motivated by our study [32].

5.7.1 Banning HTTP Scripts

Scripts fetched over HTTP are responsible for half of the vulnerabilities that we found. All of these vulnerabilities could be prevented by not allowing extensions to add HTTP scripts to their core extensions [82] or to HTTPS websites. Extensions that currently violate this restriction could be easily modified to comply by packaging the script with the extension or using a HTTPS URL. Only vulnerable extensions would be affected by the ban because any extension that uses HTTP scripts will be vulnerable to man-in-the-middle attacks.

Core Extension Vulnerabilities. Banning HTTP scripts from core extensions would remove 28 core extension vulnerabilities (56% of the total core extension vulnerabilities) from 15 extensions. These 15 extensions load HTTP scripts from 13 domains, 10 of which already offer the same script over HTTPS. The remaining 3 scripts are static files that could be packaged with the extensions.

Website Vulnerabilities. Preventing extensions from adding HTTP scripts to HTTPS websites would remove 8 website vulnerabilities from 8 extensions (46% of the total website vulnerabilities). These vulnerabilities allow a network attacker to circumvent the protection that HTTPS provides for websites. The extensions load HTTP scripts from 7 domains, 3 of which offer HTTPS. The remaining 4 scripts are static scripts that could be packaged with the extensions.

5.7.2 Banning Inline Scripts

Untrusted data should not be added to pages as HTML because it can contain inline scripts (e.g., inline event handlers, links with embedded JavaScript, and `<script>` tags). For example, untrusted data could contain an image tag with an inline event handler: `<img onload="doEvil();" ...>`. We find that 40% of the core extension vulnerabilities are caused by adding untrusted data to pages as HTML. These vulnerabilities could be prevented by not allowing any inline scripts to execute: the untrusted data will still be present as HTML, but it would be static. JavaScript would only run on a page if it is in a separate `.js` file that is stored locally or loaded from a trusted server that the developer has whitelisted.

Banning inline scripts from extension HTML would eliminate 20 vulnerabilities from 15 extensions. All of these vulnerabilities are core extension vulnerabilities. Content script vulnerabilities cannot be caused by inline scripts, and we cannot prevent extensions from adding inline scripts to HTTPS websites because existing enforcement mechanisms cannot differentiate between a website's own inline scripts and extension-added scripts.

However, banning inline scripts has costs. Developers use legitimate inline scripts for several reasons, such as to define event handlers. In order to maintain functionality despite the ban, all extensions would need to delete their inline scripts from HTML and move them to separate `.js` files. Inline event handlers (e.g., `onclick`) cannot simply be copied and pasted; they need to be rewritten as programmatically using the DOM API.

We reviewed the 100 extensions to determine what changes would be needed to comply with a ban on inline scripts. Applying this ban breaks 79% of the extensions. However, all of the extensions could be retrofitted to work without inline scripts without significant changes to the extension. Most of the compatibility costs pertain to moving the extensions' inline event handlers. The extensions contain an average of 7 event handlers, with a maximum of 98 and a minimum of 0 event handlers. Each of these handlers would need to be moved by the extensions' authors.

5.7.3 Banning Eval

Dynamic code generation converts strings to code, and its use can lead to vulnerabilities if the strings are untrusted data. Disallowing the use of dynamic code generation (e.g., `eval` and `setTimeout`) would eliminate three vulnerabilities: one core extension vulnerability, and two vulnerabilities that are both content script and core extension vulnerabilities.

We reviewed the 100 extensions and find that dynamic code generation is primarily used in three ways:

1. Developers sometimes pass static strings to `setTimeout` instead of functions. This coding pattern cannot be exploited. It would be easy to alter instances of this coding pattern to comply with a ban on dynamic code generation; the strings simply need to be replaced with equivalent functions.

2. Some developers use `eval` on data instead of `JSON.parse`. We identified one vulnerability that was caused by this practice. In the absence of dynamic code generation, developers could simply use the recommended `JSON.parse`.

3. Two extensions use `eval` to run user-specified scripts that extend the extensions. In both cases, their error is that they fetch the extra scripts over HTTP instead of HTTPS. For these two extensions, a ban on `eval` would prevent the vulnerabilities but irreparably break core features of the extensions.

Richards et al. present additional uses of `eval` in a large-scale study of web applications; we did not observe all of the use cases of `eval` in our set of 100 extensions [123].

We find that 32 extensions would be broken by a ban on dynamic code generation. Most instances can easily be replaced, but 2 extensions would be permanently broken. Overall, a ban on `eval` would fix three vulnerabilities at the cost of fundamentally breaking two extensions. Extensions that require `eval` could potentially be handled through a review process, similar to the review process for extensions with plugins.

5.7.4 Banning HTTP XHR

Network attacks can occur if untrusted data from an HTTP `XMLHttpRequest` is allowed to flow to a JavaScript execution sink. 30% of the 70 vulnerabilities are caused by allowing data from HTTP XHRs to execute. One potential defense is to disallow HTTP XHRs; all XHRs would have to use HTTPS. This ban would remove vulnerabilities from 17 extensions.

However, banning HTTP XHRs would have a high compatibility cost. The only way to comply with an HTTPS-only XHR policy is to ensure that the server supports HTTPS; unlike scripts, remote data cannot be packaged with extensions. Developers who do not control the servers that their extensions interact with will not be able to adapt their extensions. Extension developers who also control the domains may be able to add support for HTTPS, although this can be a prohibitively expensive and difficult process for a novice developer.

We reviewed the 100 extensions and found that 29% currently make HTTP XHRs. All of these would need to be changed to use HTTPS XHRs. However, not all of the domains offer HTTPS. Ten extensions request data from at least one HTTP-only domain. Additionally, four extensions make HTTP XHRs to an unlimited number of domains based on URLs provided by the user; these extensions would have permanently reduced functionality. For example, *Web Developer* lets users check whether a website is valid HTML. It fetches the user-specified website with an XHR and then validates it. Under a ban on HTTP XHRs, the extension would not be able to validate HTTP websites. In total, 14% of extensions would have some functionality permanently disabled.

5.7.5 Recommendations

Table 5.6 summarizes the benefits and costs of the defenses. If the set of 100 extensions were subject to all four bans, only 5 vulnerable extensions would remain, and 16 extensions would be permanently broken. Based on this evaluation, we conclude:

- We strongly recommend banning HTTP scripts and inline scripts; together, they would prevent 47 of the 50 core extension vulnerabilities, and no extension would be permanently broken. The developer effort required to comply with these restrictions is modest.
- Banning `eval` would have a neutral effect: neither the security benefits nor the costs are large. Consequently, we advise against banning `eval`.
- We do not recommend banning HTTP XHRs, given the number of extensions that would be permanently disabled by the ban. Of the 20 vulnerabilities that the ban on HTTP XHRs would prevent, 70% could also be prevented by banning inline scripts. We do not feel that the ban on HTTP XHRs adds enough value to justify breaking 14% of extensions.

Starting with Chrome 18, extensions will be subject to a CSP that enforces some of these bans [4]. Our study partially motivated their decision to adopt the bans [32], although the policy that they adopted is slightly stricter than our recommendations. The mandatory policy in Chrome 18 will ban HTTP scripts in core extensions, inline scripts, and dynamic code generation. Due to technical limitations, they are not adopting a ban on adding HTTP scripts to HTTPS websites. The policy will remove all of the core extension vulnerabilities that we found. The only extensions that the policy will permanently break are the two extensions that rely on `eval`.

5.8 Related Work

Extension Vulnerabilities. To our knowledge, our work is the first to evaluate the efficacy of the Google Chrome extension platform, which is widely deployed and explicitly designed to prevent and mitigate extension vulnerabilities. Vulnerabilities in other extension platforms, such as Firefox, have been investigated by previous researchers [104, 27]. We found that 40% of Google Chrome extensions are vulnerable, which is in contrast to a previous study that found that 0.24% of Firefox extensions contain vulnerabilities [27]. This does not necessarily imply that Firefox extensions are more secure; rather, our scopes and methodologies differ. Unlike the previous study, we considered network attackers as well as web attackers. We find that 5% of Google Chrome extensions have the types of web vulnerabilities that the previous study covered. The remaining discrepancy could be accounted for by our methodology: we employed expert human reviewers whereas previous work relied on a static analysis tool that does not model dynamic code evaluation, data flow through the extension API, data flow through DOM APIs, or click injection attacks. However, it is impossible to know precisely what percentage of the discrepancy is due to differences in methodology.

Privilege Separation. Privilege separation is a fundamental software engineering principle proposed by Saltzer and Schroeder [127]. Numerous works have applied this concept to security, such as OpenSSH [122] and qmail [34]. Studies have established that privilege separation has value in software projects that employ security experts (e.g., browsers [55]). However, we focus on the effectiveness of privilege separation in applications that are not written by security experts.

Recently, researchers have built several tools and frameworks to help developers privilege separate their applications [37, 65, 93, 94, 113]. For example, Akhawe et al. demonstrated how to privilege-separate Chrome extensions using HTML5 features [23]. These frameworks are targeted at security-conscious developers who voluntarily choose to privilege-separate their applications; in contrast, we consider the role of privilege separation in arbitrary applications, many of which are built by non-security-conscious developers.

In concurrent and independent work, Karim et al. studied the effectiveness of privilege separation in Mozilla Jetpack extensions [88]. Like Chrome extensions, Jetpack extensions are split into multiple components with different permissions. They statically analyzed Jetpack extensions and found several capability leaks in modules. Although none of these capability leaks are actual vulnerabilities, the capability leaks demonstrate that developers can make errors in a privilege-separated environment. Their findings support the results of our analysis of privilege separation in Chrome extensions.

CSP Compatibility. Adapting websites to work with CSP can be a challenging undertaking for developers, primarily due to the complexities associated with server-side templating languages [152]. However, extensions do not use templating languages. Consequently, applying CSP to extensions is easier than applying it to websites in most cases. We expect that our CSP compatibility findings for extensions will translate to packaged JavaScript and packaged web applications.

Malicious Extensions. Extension platforms can be used to build malware (e.g., FFsniFF and Infostealer.Snifula [157]). Mozilla and Google employ several strategies to prevent malicious extensions, such as domain verification, fees, and security reviews. Liu et al. propose changes to Chrome to make malware easier to identify, such as making permissions even more fine-grained [103]. Research on extension malware is orthogonal to our work, which focuses on external attackers that leverage vulnerabilities in benign-but-buggy extensions.

5.9 Conclusion

We performed a security review on a set of 100 Google Chrome extensions, including the 50 most popular, and found that 40% have at least one vulnerability. Based on this set of vulnerabilities, we evaluated the effectiveness of Chrome's three extension security mechanisms: isolated worlds, privilege separation, and permissions.

We found that the isolated worlds mechanism is highly effective because it prevents common developer errors (i.e., data-as-HTML errors). The effectiveness of isolated worlds means that privilege separation is rarely needed. Privilege separation's infrequent usefulness may not justify the complexity and communication overhead that it adds to extensions. However, our study shows

that privilege separation would improve security in the absence of isolated worlds. We also found that permissions can have a significant positive impact on system security; developers of vulnerable extensions can use permissions well enough to reduce the scope of their vulnerabilities.

Although we demonstrated that privilege separation and permissions can mitigate vulnerabilities, developers do not always use them optimally. We identified several instances in which developers accidentally negated the benefits of privilege separation or intentionally circumvented the privilege separation boundary to implement features. Similarly, extensions sometimes ask for more permissions than they need (Chapter 4). Automated tools for privilege separation and permission assignment could help developers better use these security mechanisms, thereby rendering them even more effective.

Despite the successes of these security mechanisms, extensions are widely vulnerable. The vulnerabilities occur because the system was designed to address only one threat: websites that attack extensions through direct interaction. There are no security mechanisms to prevent direct network attacks on core extensions, website metadata attacks, or attacks on websites that have been altered by extensions. This finding should serve as a reminder that multiple threats should be considered when initially designing a system. We propose to prevent these additional threats by banning insecure coding practices that commonly lead to vulnerabilities; bans on HTTP scripts and inline scripts would remove 94% of the most serious attacks with a tractable developer cost.

5.10 Acknowledgements

We thank Nicholas Carlini for reviewing the extensions. We also thank Prateek Saxena and Adam Barth for their insightful comments.

Chapter 6

Android Permissions: User Attention, Comprehension, and Behavior

6.1 Introduction

The purpose of Android permissions is to "inform the user of the capabilities [their] applications have" [24]. In this chapter, we explore whether Android permissions are usable security indicators that fulfill this goal. We base our inquiry on Wogalter's Communication-Human Information Processing (C-HIP) model, which provides a framework for structuring warning research [155]. The C-HIP model identifies a set of steps between the delivery of a warning and the user's final behavior. We connect each step with a research question:

1. *Attention switch and maintenance*. Do users notice permissions before installing an application? A user needs to switch focus from the primary task (i.e., installation) to the permission warnings, for long enough to read and evaluate them.
2. *Comprehension and memory*. Do users understand how permissions correspond to application risks? Users need to understand the scope and implications of permissions.
3. *Attitudes and belief.* Do users believe that permissions accurately convey risk? Do users trust the permission system to limit applications' abilities?
4. *Motivation.* Are users motivated to consider permissions? Do users care about their phones' privacy and security, or view applications as threats?
5. *Behavior.* Do permissions influence users' installation decisions? Do users ever cancel installation because of permissions? Users should not install applications whose permissions exceed their comfort thresholds.

We focus on the first two steps because they are the most important: a failure of usability in one of these early steps will render all subsequent steps irrelevant. We also study the end behavior, for an end-to-end assessment of how Android permissions affect user actions.

We performed two usability studies to address the attention, comprehension, and behavior questions. First, we surveyed 308 Android users with an Internet questionnaire to collect data about their understanding and use of permissions. Next, we observed and interviewed 25 Android users in a laboratory study to gather nuanced data. The two studies — performed with different sets of participants and in different contexts — serve to confirm and validate each other.

Our primary findings are:

- *Attention*. In both the Internet survey and laboratory study, only 17% of participants paid attention to permissions during a given installation. At the same time, 42% of laboratory participants were unaware of the existence of permissions.
- *Comprehension.* Overall, participants demonstrated very low rates of comprehension. Only 3% of Internet survey respondents could correctly answer three comprehension questions. However, 24% of laboratory study participants demonstrated a competent—albeit imperfect—understanding of permissions.
- *Behavior.* A majority of Internet survey respondents claimed to have decided not to install an application because of its permissions at least once. Twenty percent of our laboratory study participants were able to provide concrete details about times that permissions caused them to cancel installation.

Our findings indicate that the Android permission system is neither a total success nor a complete failure. Due to low attention and comprehension rates, permissions alone do not protect most users from undesirable applications (i.e., malware or grayware). However, a minority of laboratory study participants (20%) demonstrated awareness of permissions and reasonable rates of understanding (comprehension grades of 70% or higher). This minority could be sufficient to protect others if their opinions about application permissions could be successfully communicated via user reviews. We also found that some people have altered their behavior based on permissions, which demonstrates that users can be receptive to security and privacy warnings during installation.

6.2 Methodology

We surveyed 308 Android users with an Internet survey and interviewed 25 Android users in a laboratory study. We designed the two studies to validate each other. We recruited Internet survey respondents with AdMob advertisements and laboratory study participants with Craigslist advertisements; although both recruitment procedures might introduce bias, it is unlikely that they introduced the same biases. We piloted our studies with 50 AdMob-recruited Internet respondents and interviews of acquaintances.

6.2.1 Internet Survey

In September 2011, we recruited Android users to answer an Internet survey about Android permissions. The purpose of this survey was to gauge how widely users understand and consider Android permissions. To recruit respondents, we commissioned an advertising campaign using AdMob's Android advertising service. Our advertisement was displayed in applications on Android devices in the U.S. and Canada. (The advertisement did not appear on web sites.) As an incentive to participate, each person who completed a survey received a free MP3 download from Amazon.com. The advertisement included our university's name and said, "Survey for free Amazon MP3." We recruited people with AdMob advertisements because doing so restricted survey respondents to those using applications on Android devices.

We paid AdMob $\$0.116$ per click and received $31,984$ visitors, of which $1,994$ (1%) began and 350 (17.5%) completed the survey. The rate at which people began the survey was likely influenced by the high rate of accidental clicks on advertisements on mobile devices [16] and our request that only people age 18 and over take the survey. Among people who started the survey, the completion rate was likely influenced by the difficulty of completing a survey on a phone. We ran the advertisement for two hours, and respondents completed it in an average of seven minutes.

We filtered out respondents who (1) stated that they were under 18, (2) had non-Android `user-agent` strings, or (3) appeared to be duplicates based on their IP addresses and `user-agent` strings. This left us with 326 unique responses. We designed our survey to make cheating (i.e., false responses for the purpose of receiving the reward) easy and obvious by making every question optional and providing an "I don't know" option for each question. Survey responses fell into two distinct groups: responses that were complete except for two or three "I don't know" responses, and responses that were incomplete except for one or two completed questions. Thus, we filtered out responses in the latter group. This resulted in a total of 308 valid responses.

The 308 respondents reported that they were 50% male and 49% female, with the remainder declining to report their gender. Respondents indicated that their age distribution was: 28% between the ages of 18 and 28, 28% between the ages of 29 and 39, 22% between the ages of 40 and 50, 15% between the ages of 51 and 61, and 5% over the age of 62. This age distribution is in line with Android age demographics [22], although the gender breakdown of our survey is more balanced than overall Android demographics.

The survey was nine pages long and meant to be completed on an Android smartphone. Each page filled a standard phone screen. We used the first three pages to ask respondents about Android usage information: how long they had owned an Android phone, from where they had downloaded Android applications, and the factors they considered when downloading applications. On each of the three subsequent pages, we randomly displayed 1 of 11 Android permission warnings and asked respondents to indicate what the permission allows the application to do. We gave respondents four choices, in addition to "none of these" and "I don't know." We then asked respondents to complete the three Westin index questions,[1] tell us about their past actions relating to Android permissions, and provide demographics information (age and gender).

[1]The Westin index is a set of three questions designed to segment users into three groups: Privacy Fundamentalists, Privacy Pragmatists, and Privacy Unconcerned [142]. The Westin index is widely used in surveys to gauge users'

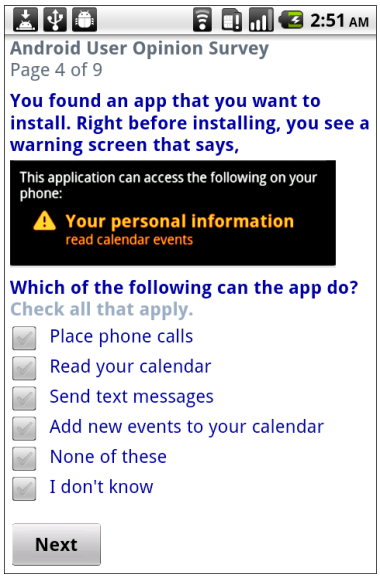


Figure 6.1. Screenshot of a quiz question from the Internet survey.

Figure 6.1 depicts one of the quiz questions from the survey, and Table 6.3 lists the 11 quiz questions and choices. We designed the permission quiz questions to include one completely incorrect choice and one choice to test fine-grained comprehension (e.g., whether they understood that a permission to read calendar events does not include the privilege to edit the calendar). The set of 11 quiz questions included two questions about the READ_SMS permission: one to test the distinction between reading and sending SMS messages, and another to test respondents' familiarity with the "SMS" acronym. Survey respondents received only one of these two related questions, so scores for these questions were independent of each other.[2]

All of the quiz questions had one or two correct choices, with the exception of the question about the CAMERA permission. This permission controls the ability to take a new photograph or video recording; it does not control access to the photo library. However, we later discovered that all applications can view or edit the photo library without any permission. Consequently, the correct answer to the CAMERA permission question is to select all four choices.

6.2.2 Laboratory Study

In October 2011, we recruited 25 local Android users for a laboratory study. The primary purpose of the laboratory study was to supplement the Internet survey with detailed and explanatory data. Technology users' feelings about privacy are complicated and often contradictory. When asked directly about their privacy preferences, most surveys have found that people are very protective of

attitudes towards privacy [97]. Buchanan et al. validated the Westin index for use in a computing context by showing that it correlates with users' privacy concerns and behavior on the Internet [45].

[2]We refer to these questions as $\texttt{READ\_SMS}_1$ and $\texttt{READ\_SMS}_2$, as depicted in Table 6.3.

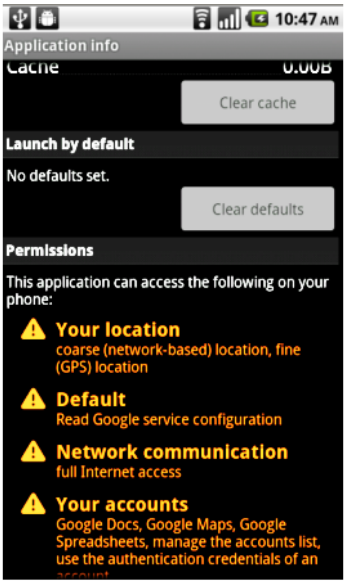


Figure 6.2. Screenshot of permissions on an application's Settings page.

their personal data [18, 45]. However, users' actions do not always correspond to their professed preferences [84]. This may be because users overestimate their privacy concerns or do not understand the ramifications of their actions (i.e., the user does not understand that the action violates his or her privacy preferences). As such, we design our laboratory study to be robust to over-reporting of security concerns by directly observing participants and asking questions about participants' past actions.

To recruit participants, we posted a Craigslist ad for the San Francisco Bay Area. Our advertisement offered people $60 to participate in an hour-long interview about how they "choose and use Android applications." In order to be eligible for the laboratory study, we required that participants owned an Android phone and used applications. We also asked study applicants to look at a screenshot and tell us whether they had the new or old version of the Android Market; we then secretly limited eligibility to users with the newer version of the Android Market. Google released a new version of the Market in August 2011, and not all phones had been upgraded yet. We decided to focus on users with the new version of the Market to reduce study variability.

Our Craigslist advertisement yielded 112 eligible participants. In order to match our participants' ages to Android demographics [22], we grouped applicants by age and selected a random proportion of people from each age group. We scheduled interviews with 30 participants. Three people failed to attend and two people had technical problems with their phones, leaving us with 25 completed interviews (12 women and 13 men). The age distribution was close to overall Android age demographics by design, with 20% of participants between 18 and 24, 32% between 25 and 34, 20% between 35 and 44, 16% between 45 and 54, and 12% older than 55. None of the participants were affiliated with our institution, although some of the younger participants were students at other universities.

Each interview took 30–60 minutes and had six parts:

1. General Android usage questions (e.g., how many applications they have installed).

2. Participants were instructed to find and install an application from the Android Market, using their own phones. We prompted them to install a "parking finder app that will help [the user] locate your parked car." This task served to confirm that participants were familiar with installing applications from the Android Market.

3. Participants were instructed to find and install a second application from the Android Market using their own phones. We prompted them to:

   > *Pretend you are a little short on cash, so you want to install a coupons app. You want to be able to find coupons and sales for groceries, your favorite electronics, or clothes while you're out shopping. If you already have a coupons app, pretend you don't like it and want a new one.*

   All of the top-ranked applications for search terms related to this scenario had multiple permissions. During this application search process, we asked participants to tell us what they were thinking about while using the Market. We also observed what user interface elements they interacted with.

4. Westin index questions.[1]

5. We asked participants about an application on their phone that they had installed and recently used. We then opened the application's information page in Settings (Figure 6.2) and asked them to describe and explain the permissions.

6. We asked participants for specific details about past permission-related behaviors, such as whether they have ever looked up permissions or decided not to install an application because of its permissions.

Two researchers performed each interview, with one acting as the interviewer and the other acting as a notetaker. To promote a casual atmosphere, we held the interviews at a coffee shop and offered participants coffee, tea, or water. Participants used their own phones to encourage them to behave as they would in the real world. We made an effort to not prime participants to security or privacy concerns until the fourth task, at which point we specifically asked them about their attitudes towards privacy. We introduced ourselves as computer science students and did not reveal that we were security researchers until the end of the study. We prevented participants from determining the security focus of the study in advance by posting the Craigslist advertisement in the name of a researcher with no online presence or prior publications.

6.3 Attention During Installation

Do users notice Android permissions before installing an application? Attention is a prerequisite for an effective security indicator: a user cannot heed a warning that he or she does not notice. In our Internet survey we asked respondents whether they looked at permissions during installation.

To supplement this self-reported statistic, we empirically determined whether laboratory study participants were aware of permission warnings. We also report users' attention to user reviews, which are shown during installation.

6.3.1 Permissions

Internet Survey

In our Internet survey, we asked respondents, "The last time you downloaded an Android application, what did you look at before deciding to download it?" Respondents were able to select multiple choices from a set of options that included "Market reviews," "Internet reviews," "screenshots," and "permissions."

We found that 17.5% of our 308 respondents (95%CI: [13.5%, 22.3%]) reported looking at permissions during their last application installation. Respondents who can be classified as Privacy Fundamentalists using the Westin index were significantly more likely to report looking at permissions than other respondents ($p < 0.0005$; Fisher's exact test). While statistically significant, the proportion of Privacy Fundamentalists who claimed to look at permissions was still a minority: 40.5% of the 42 Privacy Fundamentalists reported looking at permissions, whereas 13.9% of the remaining 266 respondents reported looking at permissions.

This self-reported question suffers from two limitations: some people over-report security concerns, and others may read permissions without knowing the technical term that refers to them. We asked survey respondents specifically about their "last installation" to discourage over-reporting, but people may still guess when they cannot remember. Our laboratory study served to confirm the results of the survey on a second population with a different metric.

Laboratory Study

In the follow-up laboratory study, we performed an experiment to empirically determine whether users noticed permissions during installation. We instructed study participants to talk us through the process of searching for and installing a coupon application. We recorded whether they clicked on or mentioned the permissions on the final Market installation page. To avoid priming participants, we did not mention permissions unless the participant verbally indicated that he or she was reading them. After each participant passed through the page with permissions, we asked him or her to describe what had been on the previous page.

We categorized participants into three groups:

Attention to Permissions	Number of users		95% CI
Looked at the permissions	4	17%	5% to 37%
Didn't look, but aware	10	42%	22% to 63%
Is unaware of permissions	10	42%	22% to 63%

Table 6.1. Attention to permissions at installation (Lab Study, $n = 24$)

- *Participants who looked at permissions during the installation*. These participants either told us that they were looking at permissions while on the page with permissions or they were later able to provide specific details about the contents of that page. They were also able to discuss permissions in general, indicating that the laboratory study was not the first time that they had viewed permissions. For example, one participant opened the page with permissions and stated,

  > *The only thing I started doing recently, is kinda looking at these — is there anything really weird.*

  When questioned, that participant described concern over "the network stuff."

- *Participants who did not look at the permissions for this specific application, but were able to tell us that the final installation page listed permissions*. In order to answer our question, these participants must have paid attention to permissions at some point in the past. For example, one participant in this category responded,

  > *I've seen a lot of them...A lot of 'em have full network access, access to your dialer, your call logs, and GPS location also.*

- *Participants who were unaware that the final installation page included a list of permissions*. For example, one participant said, "I don't remember. I just remember 'Download and install'." Another said, "I don't ever pay attention. I just accept and download it."

We did not require knowledge of the term "permissions"; participants typically used other phrases (e.g., "little warning things") to describe what they saw or remembered.

Table 6.1 shows the number of study participants who fell into each of the three categories. Fourteen participants (58% of 24) noticed permissions during the experimental installation or reported paying attention to permissions in the past.[3] The remaining participants were unaware of the presence of permissions on the final installation page in the Market. We did not observe a relationship between Westin indices and participants' attention to permissions.

Of the ten participants who did not look at permissions during the study but were aware of them, three volunteered that they used to look at permissions but no longer do. For example, one participant said, "I used to look...I just stopped doing that." These participants might have experienced warning fatigue, since users see permission warnings for about 90% of applications (Chapter 4).

[3]For this statistic, we omit one participant who had never previously completed an installation without help.

One participant said that she used to be concerned about the location permission, but gradually lost her concern because so many of the applications that she installed requested this permission.

Of the ten participants who had never paid attention to permissions, two knew that they were accepting an agreement on the final installation page. They both described the page as containing legal terms of use, with one incorrectly elaborating that the text specified legal restrictions on the use of the application. Due to their lack of interest in legal text, neither had ever read the screen so they were unaware that the text pertains to security and privacy.

The self-reported survey and observational study results both suggest that 17% of users routinely look at permissions when installing an application. We also found that 42% of study participants could not possibly benefit from permission information because they had never noticed it. The remaining 42% of participants were aware of permissions but do not always consider them.

6.3.2 Reviews

Like permissions, user reviews have the ability to convey privacy and security information during installation. User reviews can warn people about undesirable or privacy-invasive applications.

Internet Survey

We asked survey respondents, "The last time you downloaded an Android application, what did you look at before deciding to download it?" A total of 219 survey respondents (71.1% of 308; 95%CI: [65.5%, 76.2%]) reported looking at some type of review before installation. Of these, 193 respondents (62.7% of 308; 95%CI: [57.0%, 68.1%]) indicated that they looked at Market reviews during their last application installation, and 42 respondents (13.6% of 308; 95%CI: [10.0%, 18.0%]) stated that they had looked at other reviews on the Internet. Twenty-six respondents (8.4% of 308; 95%CI: [5.6%, 12.1%]) reported that they had looked at both Internet and Market reviews. We did not find that any age, gender, or Westin group was more or less likely to look at reviews.

Laboratory Study

In our follow-up laboratory study, we observed whether participants actually considered reviews during application installation. We instructed participants to tell us what they were reading and considering while selecting and installing a coupon application. We did not mention reviews or ratings unless the participant first spoke of or clicked on them. After participants mentioned reviews or ratings, we asked them how much importance they placed on reviews and whether they trusted them to be correct. If a participant did not consider reviews during installation, we asked the participant for his or her opinion of reviews after the installation task.

Importance	Read reviews	Didn't read reviews
A lot	68%	4%
Somewhat	16%	4%
Mistrust	4%	0%
Unknown	0%	4%
Total	88%	12%

Table 6.2. We observed whether users read reviews, and later asked how much importance they place on reviews (Lab Study, $n = 25$)

Table 6.2 shows participants' opinions of reviews and whether they considered reviews during the installation. All but three participants mentioned application reviews during installation; of the three that did not read reviews, two later claimed when questioned that they read reviews in some situations. The majority of participants placed a lot of importance on reviews. For example,

> *[Reviews] let me know if it's a decent app or not. Because most people will put on there whether it's a good app or a bad app.*

A few participants reported that they read reviews but simply treated them as one factor among many, rather than using them as their primary decision-making factor. One of these participants described the rating system as "a starting point," and another said that reviews are "just a place to start." One of the 25 participants actively mistrusts positive reviews because she has written reviews for her company's products on websites. Despite this, she still looks at reviews to identify negative traits of applications.

At the end of the study, we asked participants whether they had ever tried to find out what a permission means or why an application was asking for it. Eight of the study participants (32% of 25) responded affirmatively, with six (24% of 25) people stating that they found this information in some type of review. Three of these participants stated that they had read user reviews to determine whether an application's permissions were appropriate, and another two said that they had read news articles that reviewed applications' permissions. Another participant said that he read about permission information in reviews, but that he had never noticed that the same permission information was available on the final installation page. One of the six said,

> *If I'm not sure about an app I'll research it and see what other people say about the permissions. Like, 'It does this,' ...and, 'It's necessary.' And there will be an argument or a discourse about the permissions that need to be on there or don't need to be.*

This suggests that reviews are an important part of communicating permission information, especially for users who do not understand permission warnings on their own.

Permission	*n*	Options	Responses	
INTERNET Category: Network communication Label: Full Internet access	109	✔ Send information to the application's server	45	41.3%
		✔ Load advertisements	30	27.5%
		✗ None of these	16	14.7%
		✗ Read your text messages	13	11.9%
		✗ Read your list of phone contacts	11	10.1%
		I don't know	36	33.0%
READ_PHONE_STATE Category: Phone calls Label: Read phone state and identity	85	✔ Read your phone number	41	47.7%
		✗ See who you have called	37	43.0%
		✔ Track you across applications	20	23.3%
		✗ Load advertisements	11	12.8%
		✗ None of these	10	11.6%
		I don't know	15	17.4%
CALL_PHONE Category: Services that cost you money Label: Directly call phone numbers	83	✔ Place phone calls	30	35.3%
		✗ Charge purchases to your credit card	27	31.8%
		✗ None of these	16	18.8%
		✗ See who you have made calls to	14	16.5%
		✗ Send text messages	11	12.9%
		I don't know	16	18.8%
WRITE_EXTERNAL_STORAGE Category: Storage Label: Modify/delete SD card contents	92	✔ Read other applications' files on the SD card	41	44.6%
		✔ Change other applications' files on the SD card	39	42.4%
		✗ None of these	16	17.4%
		✗ See who you have made phone calls to	15	16.3%
		✗ Send text messages	11	12.0%
		I don't know	15	16.3%
WAKE_LOCK Category: System tools Label: Prevent phone from sleeping	81	✔ Keep your phone's screen on all the time	49	60.5%
		✔ Drain your phone's battery	37	45.7%
		✗ None of these	7	8.6%
		✗ Send text messages	4	4.9%
		✗ Delete your list of contacts	4	4.9%
		I don't know	13	16.0%
CHANGE_NETWORK_STATE Category: System tools Label: Change network connectivity	66	✔ Turn your WiFi on or off	36	52.9%
		✗ Send information to the application's server	13	19.1%
		✗ Read your calendar	7	10.3%
		✗ None of these	7	10.3%
		✗ See who you have made calls to	5	7.4%
		I don't know	17	25.0%
READ_SMS$_2$ Category: Your messages Label: Read SMS or MMS	54	✔ Read text messages you've sent	30	54.5%
		✔ Read text messages you've received	25	45.5%
		✗ Send text messages	10	18.2%
		✗ Read your phone's unique ID	6	10.9%
		✗ None of these	4	7.3%
		I don't know	11	20.0%
READ_SMS$_1$ Category: Your messages Label: Read SMS or MMS	77	✔ Read text messages you've received	44	56.4%
		✗ Read e-mail messages you've received	30	38.5%
		✗ Read your call history	13	16.7%
		✗ None of these	8	10.3%
		✗ Access your voicemail	8	10.3%
		I don't know	13	16.7%
READ_CALENDAR Category: Your personal information Label: Read calendar events	101	✔ Read your calendar	56	53.3%
		✗ None of these	18	17.1%
		✗ Add new events to your calendar	12	11.4%
		✗ Send text messages	12	11.4%
		✗ Place phone calls	9	8.6%
		I don't know	19	18.1%
READ_CONTACTS Category: Your personal information Label: Read contact data	86	✔ Read your list of contacts	52	60.5%
		✔ Read your call history	19	22.1%
		✗ None of these	14	16.3%
		✗ Delete your list of contacts	9	10.5%
		✗ Place phone calls	5	5.8%
		I don't know	14	16.3%
CAMERA Category: Hardware controls Label: Take pictures	72	✔ Take pictures when you press the button	27	37.0%
		✔ Take pictures at any time	27	37.0%
		✔ See pictures taken by other applications	16	21.9%
		✔ Delete pictures taken by other apps	13	17.8%
		✗ None of these	13	17.8%
		I don't know	17	23.3%

Table 6.3. Survey respondents were each asked three questions, randomly selected from this set. Respondents could select "None," "I don't know," or one or more of the four definitional choices. This table orders the choices by response rate. Checks are correct answers, and X-marks are incorrect answers.

6.4 Comprehension of Permissions

Do users understand how permissions correspond to application privileges? Users can only make correct security decisions based on permissions if they understand what the permission warnings mean. We used three metrics to measure subjects' understanding of permission warnings. First, we tested Internet survey respondents with multiple-choice questions (Chapter 6.4.1). Second, we graded laboratory study participants' ability to describe the permission warnings of a familiar application (Chapter 6.4.2). Third, we asked study participants whether the application's set of permissions gave it the ability to send text messages (Chapter 6.4.3).

6.4.1 Permission Comprehension Quiz

Internet survey respondents answered three randomly-selected quiz questions from the set of eleven questions in Table 6.3. Six respondents omitted one or more questions; we filtered those participants out of this analysis, leaving us with 302 respondents who answered three quiz questions. Overall, respondents did not randomly select their responses: the observed median differs from $1/16$, which would be the median if respondents were equally likely to select any choice or set of choices (one-sample Wilcoxon Signed Rank test, $p < 0.0005$).

Eight respondents (2.6% of 302) answered all three questions correctly. On average, respondents correctly answered 21% of the three questions. We considered the relationship between respondent scores and demographics:

- We did not observe a correlation between respondent scores and the length of Android phone ownership.
- No significant differences were observed between the genders or with regard to Westin index classifications.
- There was a negative correlation between age and the number of correct answers ($r = -0.257$, $p < 0.0005$); younger people were more likely to understand permissions.
- We compared the scores of respondents who did and did not report looking at permissions in a past application installation. Respondents who reported looking at permissions scored higher on average (30.3% vs. 18.6%). The difference was statistically significant ($U = 5,293.0$, $p < 0.007$, $r = 0.16$) but small in absolute terms.
- In the survey, we asked respondents whether they typically used the Android Market or unofficial application stores. Respondents who typically used the Android Market were significantly more likely to understand the permissions ($U = 2,474.0$, $p < 0.001$, $r = 0.20$). The 28 respondents who did not use the Market had an average score of 4.7%, whereas the remaining 274 respondents had an average score of 22.3%.

	Permission	n	Correct Answers	
1 Choice	READ_CALENDAR	101	46	45.5%
	CHANGE_NETWORK_STATE	66	26	39.4%
	READ_SMS$_1$	77	24	31.2%
	CALL_PHONE	83	16	19.3%
2 Choices	WAKE_LOCK	81	27	33.3%
	WRITE_EXTERNAL_STORAGE	92	14	15.2%
	READ_CONTACTS	86	11	12.8%
	INTERNET	109	12	11.0%
	READ_PHONE_STATE	85	4	4.7%
	READ_SMS$_2$	54	12	22.2%
4	CAMERA	72	7	9.7%

Table 6.4. The number of people who correctly answered a question. Questions are grouped by the number of correct choices. n is the number of respondents. (Internet Survey)

Although we found statistically significant differences between certain groups, no group performed well on an absolute scale.

Despite their poor overall scores, many respondents demonstrate a limited understanding of the permission warnings. As shown in Table 6.3, a plurality of respondents selected at least one correct choice for every question, and a majority of respondents selected at least one correct choice for six questions. Respondents' low scores reflect the fact that we only consider an answer correct if that respondent specified all of the correct choices and no incorrect choices. Although only 21% of the 906 answers[4] were completely correct, 53% of the answers contain at least one correct choice: 17% of answers contain a subset of the correct choices, and 15% of answers contain extra incorrect choices. This type of error indicates that the respondent can identify (part of) the definition, but he or she incorrectly believes that the permission's scope is narrower or broader than it actually is.

Seven questions had multiple correct choices. Users performed significantly worse on questions with multiple correct choices ($r = -0.59$, $p < 0.028$; one-tailed), so we can only directly compare permissions with the same number of possible correct choices. Table 6.4 depicts the eleven permissions and the number of survey respondents who got each one completely correct.

We hypothesize that some respondents made decisions based primarily on the category headings, which are featured in a much larger font than the specific permission labels. This may have led respondents to overstate the meanings of permissions (i.e., they selected incorrect as well as correct choices). Respondents' answers to all but one of the permissions seem consistent with this hypothesis (Table 6.3). For example, the CALL_PHONE permission illustrates this type of error: the large category heading says "Services that cost you money," and nearly as many respondents selected the incorrect answer of "Charge purchases to your credit card" as the correct answer of "Place phone calls." The one question that does not fit this model is READ_SMS$_2$: 64% of respondents correctly determined that the READ_SMS$_2$ permission grants the ability to read but not send text messages (although many selected a subset of the correct responses).

[4]906 answers: 302 respondents provided three answers each.

6.4.2 Free-Form Permission Descriptions

We hypothesized that users might understand permission warnings better when the permissions are associated with a familiar application. For example, a user who does not understand the `INTERNET` permission in isolation might know that the permission is needed to fetch news from the Internet when he or she sees that the permission is associated with a news application. As such, we designed our follow-up laboratory study to ask users about the meaning of permissions in the context of a familiar application.

During the laboratory study, we asked each participant to view the permissions of an application that he or she had recently used on his or her phone. The participant was therefore familiar with the application's functionality. We asked participants to read each permission aloud and explain what it meant. We gave participants three chances to demonstrate their understanding of the permissions: we asked what the permissions meant, why the application had them, and whether each permission was necessary or unnecessary for the respective application.

To evaluate user understanding, we graded participants' free-form descriptions of permissions as follows:

- *Correct.* A correct answer completely explains the meaning of a permission. For example, one participant correctly stated that the `BLUETOOTH_ADMIN` permission allowed the application to "create a Bluetooth connection" and "disconnect Bluetooth to save battery."

- *Correct but overly broad.* This type of answer contained correct information, but the participant believed that the permission granted more privileges than it actually does. For example, one participant understood that the `INTERNET` permission could be used to send or retrieve data, but he also believed it gave the application the ability to "check my GPS or see where I'm going." (In this example, the application did not have a location permission.)

- *Incomplete.* Incomplete answers show that the participant had a partial understanding of the permission, but lacked comprehension of an important aspect of the permission. For example, one participant understood that the `RECEIVE_SMS` permission pertains to text messages but did not know how.

- *Incomplete and overly broad.* This type of answer includes wrong information in addition to an incomplete but correct explanation of a permission. For example, one participant described the `READ_PHONE_STATE` permission as,

  > *Phone calls is probably like the call log or the phone calls that are made. It tells you their names and maybe a picture.*

  This description is partially correct because the permission relates to call state, but it is incomplete because the permission also provides access to the participant's own identity. The participant also incorrectly stated that the permission grants access to contacts' names and pictures.

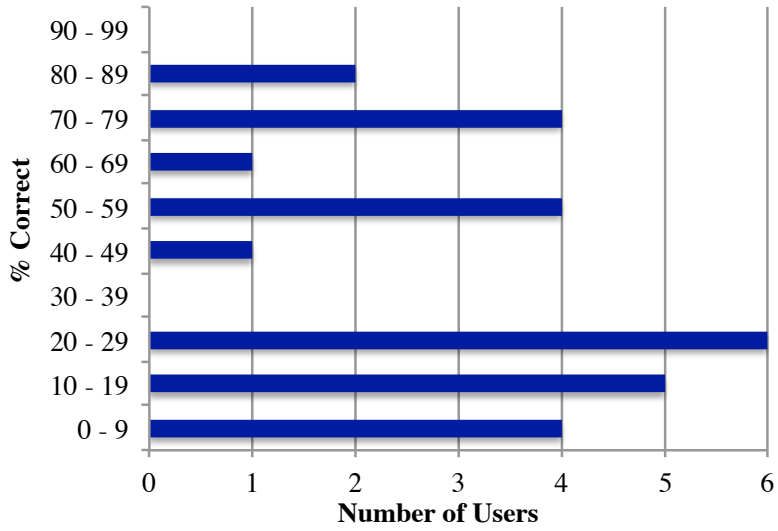


Figure 6.3. A histogram of participants' grades. (Lab Study, $n = 25$)

- *Wrong.* The participant provided an incorrect description of a permission that included substantially more privileges than the truth. For example, several participants stated that the `READ_PHONE_STATE` permission applications listen to their phone calls. This confusion likely occurred because the category heading for that permission is "Phone Calls."
- *Unable to answer.* We placed responses in this category when the participant read the permission aloud and then stated that he or she could not describe the permission.
- *Omitted.* Participants often skipped permissions that were present on the screen, and we were not always able to prompt them to address the skipped permission. In these cases, we have no way of knowing whether the participant would have been able to answer correctly.

Figure 6.3 depicts each laboratory study participant's grade, where a person's grade is the percentage of descriptions that were correct. We calculated this percentage after filtering out permissions that they omitted; omitted permissions are excluded because we do not know why they were omitted. Two participants received grades of 0%, the highest grade was 83%, and the average was 39%. Contrary to our initial hypothesis, comprehension rates are still low when permissions are associated with a familiar application.

Six participants received grades of 70% or higher. We observed two of the six high scorers looking at permissions during installation (Chapter 6.3.1), and another three indicated that they had looked at permissions in the past. The permission system could potentially help these five participants (20% of 25) because they sometimes pay attention to and understand permission warnings. The sixth high scorer expressed some familiarity with permissions but did not know that the Market displayed them prior to installation.

Other participants commonly said that they did not know what the warnings meant: 25% of the times that a participant read a permission, he or she was completely unable to describe it.

	READ_CONTACTS	WAKE_LOCK	WRITE_EXTERNAL_STORAGE	READ_PHONE STATE	INTERNET
Correct	0%	54%	47%	0%	68%
Correct but overly broad	9%	9%	0%	0%	4%
Incomplete [and overly broad]	18%	0%	18%	45%	9%
Wrong	45%	0%	23%	20%	9%
Unable to answer	27%	36%	12%	35%	9%
Total number of participants	11	11	17	20	22

Table 6.5. The grades of free-form responses for common permissions. (Lab Study, $n = 25$)

Several participants mentioned that they understood all of the vocabulary but did not know how that information pertained to their phones. As one participant said,

> *I think I know what they mean as a person who has zero electronics or programming training, just in terms of what I think the words mean...I know what they mean in terms of the face value of the words. I don't really know what they mean in terms of complicated, in terms of technicalities...*

Participants' grades are not directly comparable to each other because each participant viewed a different application's permissions. However, a few popular permissions were present in 11 or more participants' applications. Table 6.5 shows how participants performed when describing these permissions. Notably, participants performed better on the three permissions that refer to general computer concepts: Internet access, hard drive storage, and putting the phone to "sleep." Participants were less able to describe the two smartphone-specific permissions.

We observed that participants often placed more emphasis on the category heading than the specific permission text, which caused them to err in the direction of overstating the privilege associated with permissions. (Figure 3.1 shows examples of categories and specific permissions.) Descriptions were overly broad 29% of the time, and all but 3 of the overly-broad responses could be attributed to the category heading. For example, the READ_CONTACTS permission is under the heading of "Personal Information." Upon seeing that warning, one participant stated that the permission provided access to his passwords, and another believed that the permission encompassed all of the data on her phone. Similarly, the READ_PHONE_STATE permission is under the heading of "Phone Calls." Participants inferred that the warning referred to a wide variety of phone-related behavior, such as giving a company permission to make telemarketing calls to the participant.

6.4.3 Specific Permission Comprehension

After each participant described the set of permissions, we asked him or her whether the selected application had the ability to send text messages without his or her knowledge. If the participant asked for clarification, we elaborated that we wanted to know whether the application *can* send text messages, not whether it does. This privilege is granted with the SEND_SMS permission, which is in the "Services that cost the user money" category with the specific permission label of "Send SMS messages." This question was designed to gauge whether people can determine the tasks that

Participant response:	**Yes**	**No**	**Unsure**
Correct	4%	32%	-
Incorrect	44%	12%	-
Total	48%	44%	8%

Table 6.6. Can an application send text messages? The correct answer depends on the application that the given user selected. (Lab Study, $n = 25$)

an application can do on their phones, given its permissions. We chose the `SEND_SMS` permission for this question because we thought that all participants would be familiar with text messages, and the permission is associated with malware [64, 159].

Table 6.6 presents participants' responses. The correct answer depends on the application that the given user selected. Only nine of the participants (36% of 25) answered correctly. Participants' answers were not significantly different from guessing: twelve responded affirmatively and eleven negatively. None of the participants' confusion stemmed from the language of the warning; instead, their responses indicated that they did not understand the role or scope of permissions in general.

Four participants selected applications with the `SEND_SMS` permission, and three of them incorrectly stated that the application could not send text messages. One of these participants had asked us about the meaning of the `SEND_SMS` permission during the previous step of the laboratory study, and she correctly repeated our explanation. Despite this, she still responded that the application could not send text messages. She re-examined the permission warning after our question and stated,

> *Well, I don't know now. Cause it said that it could — I don't know. I'm going to say no.*

Another participant was aware that her chosen application could send text messages because she had used the application, but she still believed that it was not capable of sending text messages without her expressed approval. She seemed to believe that all applications require user approval to send text messages, regardless of the permissions. The third person looked at the category heading ("Services that cost the user money") and incorrectly decided that it referred to Internet data and phone calls but not text messages.

Twenty-one participants selected applications that do not have the `SEND_SMS` permission. Of those, eleven participants incorrectly thought that their applications could send text messages. When asked why, six participants explained that various other permissions allow this behavior. Two people said that the `INTERNET` permission (listed under the "Network communication" category heading) allows an application to send a text message. For example,

> *It has access to my network, so I assume it could send a message if it wanted to.*

Four people believed that the `READ_PHONE_STATE` permission (listed under the "Phone calls" category heading) grants the ability to send text messages. For example,

Well, yeah, because of the phone calls. Because of the phone calls, they can read the phone calls, so obviously they can.

A sixth participant believed that the application could combine the personal information, phone calls, and network communication categories together to send a text message.

One of our participants said he had a small amount of experience as an Android developer. He was among the eleven participants who incorrectly stated that an application could send text messages. When asked for an explanation,

I've done some programming but I don't know all the permissions. ... I just don't know if the permissions are so fine grained that they make texting a special permission that you have to add.

The participant then reasoned that two other permissions likely include that ability. Without knowing the full list of possible Android permissions, it is difficult for a user — even a highly experienced, technically competent user – to determine whether an application cannot perform an action. In other words, users need to know what permissions their application does not have in order to comprehend the scope of the permissions that it does have.

6.5 Influence on User Behavior

Do permissions influence users' installation decisions? Users are shown permissions on the final installation page of the Market so that they can refrain from downloading an application if they dislike its requested permissions. We asked users whether they have ever decided not to install an application because of its permissions.

6.5.1 Internet Survey

The survey asked, "Have you ever not installed an app because of permissions?" Respondents were shown the following four choices:

- Yes, I didn't like the permissions
- Yes, there were too many permissions
- No
- I don't know

A respondent could select both affirmative options. The answers were not randomly ordered.

Self-Reported Behavior	Respondents
Yes	56.7%
Didn't like permissions	*32.6%*
Too many permissions	*16.0%*
Both	*8.1%*
No	34.5%
I don't know	8.8%

Table 6.7. Respondents who claimed they did not install an application due to permissions. (Internet Survey, $n = 307$)

We received 307 responses. Table 6.7 shows the results: 56.7% of respondents (95%CI: [52.1%, 62.3%]) claimed to have decided not to install an application because of its permissions. We found that respondents who can be classified as Privacy Fundamentalists using the Westin index were more likely than other respondents to report not installing an application due to its permissions (χ^2=5.6161, $p = 0.016$): 73.8% of the 42 Privacy Fundamentalists (95%CI: [60.5%, 87.1%]) responded affirmatively, compared to 53.9% of the 265 remaining respondents (95%CI: [47.9%, 59.9%]).

The number of affirmative responses to this question may be artificially inflated because of position bias; people display a slight preference for the first choice over later choices [38]. Survey respondents viewed this question after seeing the permission quiz questions, which also may have increased their likeliness to respond affirmatively. We asked survey respondents about a past action rather than a preference to mitigate over-reporting, but people may err with a bias when they cannot remember the answer.

6.5.2 Laboratory Study

In our follow-up laboratory study, we asked participants the same question: "Have you ever not installed an app because of permissions?" However, we designed the laboratory study question to avoid over-reporting. If a person responded affirmatively, we asked for detailed information about the application and why he or she objected to the permissions. Although people often over-report their security concerns when asked abstract questions, we feel it is unlikely that a participant would fabricate specific details of his or her application installation history in an in-person interview.

Table 6.8 shows how study participants responded to this question. We asked the five affirmative participants to explain why and how often they had decided not to install certain applications based on their permissions. Here, we excerpt their concerns:

- One person decided not to install a social networking application because "with exact location then they could post that on my page or something like that."
- "At least five. I felt it was asking for too much, or it was going to do too much data, and I didn't feel comfortable."

Self-Reported Behavior	Participants	
Yes	5	20%
Probably	2	8%
No	18	72%

Table 6.8. Participants who claim they did not install an application due to permissions. The two "Probably" responses refer to participants who could not provide confirming details. (Lab Study, $n = 25$)

- One participant became alarmed after reading a Wall Street Journal article about Android applications' permissions and privacy policies [144]. "I haven't really downloaded very many apps since... And there have been a few I haven't downloaded because they asked for a bunch of accesses."
- "In the zone of maybe one out of four, roughly. Mostly most of them look fairly benign to me in terms of my concerns, but there are some of them that just look like they're overkill. I must say that in the beginning of installing apps, I — and I believe most people — are more hesitant about installing apps that reveal your location."
- Another person was aware of permissions but did not read them on his own. Instead, he would look for reviews about certain permissions pertaining to battery life. "Some of the ones that people say, 'It runs at startup,' and, 'You can't stop it,' or something like that...then I won't download it."

Two of the five participants who said that they had not installed an application because of permissions scored very poorly on the comprehension study (Chapter 6.4.2). One was unable to describe any permissions correctly, and the other described only two of seven permissions correctly. This shows that people may act on permission information even if they do not correctly understand it.

Through our attention and comprehension studies, we identified five participants who were aware of and understood permissions relatively well. Two of those participants said that they had cancelled installation due to permissions in the past. In other words, 8% of 25 participants paid attention to, understood, and previously acted on permissions. It is unclear why the other three participants who paid attention to and understood permissions have never cancelled installation because of permissions; it is possible that they lack motivation, lack trust in the permission system, or have simply not yet encountered a suspicious application.

6.6 Implications

We evaluated whether the Android permission system can help users avoid security- and privacy-invasive applications. We now assess the significance of our findings and several recommendations for improving the usability of permissions.

6.6.1 Effectiveness of Permissions

Our studies demonstrated that the majority of Android users do not pay attention to or understand permission warnings. Nearly half of the laboratory study participants were completely unaware that permission warnings are displayed in the Market. Since attention and comprehension are prerequisites for informed security decisions, our study indicates that the current Android permission system does not help most users make good security decisions.

However, we also found that permissions are effective at conveying security information to a minority of users: 20% of the laboratory study participants were aware of permissions and scored above 70% on the comprehension metric. It is possible that this is sufficient; a small fraction of expert users could write negative reviews when they encounter troubling permission requests, thereby protecting other consumers. Researchers have found that negative reviews can influence sales in other contexts [133, 160], and 24% of laboratory study participants said that they had relied on reviews or news reports to provide them with permission information.

6.6.2 Short-Term Recommendations

Our studies identified several factors that contribute to the low attention and comprehension rates. We now present a set of specific design recommendations aimed at addressing these problems.

Low-Risk Warnings. We observed evidence of users experiencing warning fatigue. Warning fatigue is exacerbated by unnecessary warnings. To avoid devaluing the warnings, we recommend that permissions without clear risks should not be shown to users. For example, the ability to connect to a Bluetooth device is unlikely to cause a user harm. Warnings that do not convey real risks teach the user that all warnings are unimportant [141, 57], and there are limits on how much information people can process when making decisions [36, 69]. Currently, some permissions are not displayed to users unless they choose to "See more" because the permissions are considered non-dangerous; we recommend that more permissions should be classified as non-dangerous (and hidden by default). We study user concerns about various risks in Chapter 7.

Absent Permissions. Our SMS comprehension study demonstrated that people cannot reason about the absence of permissions. A user cannot say with certainty that a permission does *not* encompass a privilege unless the user knows that another permission exists to address that privilege or no permission permits the action. Consequently, users overestimate the scope and risk of the permissions that are present. Currently, it is infeasible for any user to remember all of the permissions, given that Android has more than 100 permissions. We recommend coalescing or paring down the list of permission warnings to a set that is small enough for users to look up and remember with accuracy.

Categories. We find that category headings widely confused users. As Figure 3.1 shows, the final installation page uses a multi-layer user interface to convey permissions. The large category headings are short, simple, and non-technical; below them, the smaller text includes more information

about the specific permissions. Multi-layer user interfaces are intended to simultaneously satisfy novice, average, and expert users by providing subsequently more information at each layer of the user interface [135, 100]. However, the category headings are currently so broad that they cause users to overestimate the scope and risk of the requested permissions. Overestimation undermines the warning system because it causes users to believe that they are granting dangerous permissions to more applications than they are. This likely has a negative impact on the amount of attention that users pay to permissions; there is little reason to read individual permission warnings if one believes that all applications receive dangerous privileges.

We recommend re-organizing and re-naming categories to shape user expectations more appropriately. In particular, the "Personal Information" and "Phone Calls" categories misled many of the users in our studies. Although the category headings need to be re-designed, we do not recommend removing them; the categories reduce warning fatigue by decreasing the number of warnings that are shown on the screen. (E.g., a user sees only three warnings for eight permissions if the eight permissions fall into three categories.)

Risks, Not Resources. We find that many users cannot connect permission warnings to risks, even if they understand all of the technical terms in a permission warning. Currently, most of the warnings are resource-centric and value-neutral (e.g., "full Internet access" and "read phone state and identity"). Users are left to decide on their own how the resources might be used, which causes them to underestimate or overestimate the risks of permissions. It is important for warnings to clearly convey specific risks [156]. We cannot expect non-expert users to understand the relationship between resources and risks, and users cannot provide informed consent if they do not realize the risks. The long explanation dialog specifies the risks for a few permissions, but the majority lack risk information; also, we did not observe any users reading the long dialogs. We recommend that permission warnings focus wholly on risks (i.e., potential negative outcomes) instead of resources. For example, "full Internet access" could be replaced with "use your data plan." To balance the risks with benefits, developers could be given space in the UI to justify why they need the permissions.

Optional Permissions. Several researchers have suggested that users should be able to grant or deny an application's permissions individually, rather than as a bundle [112, 116]. This would give users finer-grained control over the resources that applications have access to. We do not recommend adopting this proposal until user understanding of permissions can be improved with other measures. The low comprehension rates suggest that users cannot currently make informed decisions about individual permissions. Even the users that displayed comprehension competency during the laboratory study did not receive perfect comprehension scores. As such, individual permission granting would add complexity to the user interface without increasing user control.

6.6.3 Longer-Term Recommendations

Larger changes are needed to improve the relevance of permission warnings and reach users who are currently unaware of permission warnings. We present a set of open problems and future research directions that are motivated by our studies.

Timing. Android shows users permission information during installation instead of when they are using the application. This design decision was made because "over-prompting the user causes the user to start saying 'OK' to any dialog that is shown" [24]. Indeed, many studies have shown that users click through security dialogs that are presented when the user is trying to perform a task with an application [111, 141, 131]. However, we find that the install-time permission dialog is similarly dismissed by most users. Additionally, install-time permissions lack context; unlike dialogs shown at runtime, there is no way to know what application functionality the install-time permissions correspond to. This suggests that completely new solutions that avoid dialogs, such as sensor-access widgets [80] or access-control gadgets [125], may be needed. We explore alternative permission-granting mechanisms in Chapter 8.

Reviews. We identified a small minority of "expert" users who could potentially protect others by sharing their concerns about permissions. One direction is to re-think how a system could support the sharing of privacy and security concerns. How can we incentivize writing reviews about permissions? How can we help interested users determine what applications are doing with permissions so that they can write useful reviews? How can other readers confirm claims about privacy and security? Currently, Android does not provide any way to audit an application's permission usage, although researchers have developed tools for computer scientists [62, 79]. However, users with interests in privacy and security are not necessarily computer scientists, despite some familiarity with smartphone technology; none of our "expert" users had any formal technical education, and we do not expect that they would be able to use any of the existing research tools.

Customization. We hypothesize that different users have different types of privacy and security concerns. For example, a mother told us that she worried a lot about people knowing her daughter's location via their shared phone, whereas another user said he was concerned only about whether applications will excessively drain his phone's battery. When users read permissions aloud to us for the comprehension study, they often told us (without prompting) that they did not care about certain permissions. Warnings will likely be more effective if they are relevant to users' specific concerns about applications. The challenge is to identify users' concerns without expecting all users to fill out surveys or provide feedback. It might be possible to learn which warnings are likely to be relevant to particular users, classes of users, or users generally.

6.7 Conclusion

This chapter represents a first step in understanding the effectiveness of Android permissions. Our two studies indicate that Android permissions fail to inform the majority of users. Low rates of user attention and comprehension indicate that significant work is needed to make the Android permission system widely accessible. However, a minority of users demonstrated awareness and understanding of permissions, and we found that permissions helped some users avoid privacy-invasive applications. This motivates continued effort towards the goal of usable permissions.

We identified a set of issues that are impeding awareness and comprehension. In particular, category headings are confusing, some users cannot connect resource-based warnings to risks, some users cannot reason about the absence of permissions, and some users are experiencing warning

fatigue. In Chapters 7 and 8, we pursue approaches to some of these problems: identifying and removing low-risk warnings, reducing the number of warnings that users see, and investigating new types of permission-granting mechanisms.

6.8 Acknowledgments

We thank Elizabeth Ha, Serge Egelman, Ariel Haney, and Erika Chin for their help with instrument design and/or study deployment. We also thank Angie Abbatecola for her help with survey and study logistics, such as ensuring that laboratory study participants were paid in a timely fashion.

Chapter 7

A Survey of Smartphone Users' Concerns

7.1 Introduction

In Chapter 6, we found that users do not pay attention to or understand Android's install-time permission warnings. One problem is that Android controls a large number of privileges with permissions, and the multitude of warnings is difficult for users to process and recall. We recommend that a permission system should have fewer permissions, focused on the highest risks. iOS takes the extreme, opposite approach: users are only asked to provide consent for two application permissions (reading location and sending notifications). However, iPhone users were outraged when they discovered that applications access other resources without their approval [42]. These experiences suggest that neither Android nor iOS ask users about the right permissions.

In order to guide the future selection of permission warnings, we performed two surveys to rank the level of user concern about a wide range of smartphone resources. In our first survey (Chapter 7.2), we asked 3,115 smartphone users to rate their level of concern about 99 risks corresponding to 54 smartphone permissions. We ranked the risks using the percentage of respondents who said they would be "very upset" if the risks occurred. In our second survey (Chapter 7.3), we asked 42 smartphone users to state their reactions to low-ranked, medium-ranked, and high-ranked risks in their own words. The open-ended responses validate the ranking: participants viewed low-ranked risks as manageable annoyances, whereas they viewed high-ranked risks as severe offenses that may require the help of their service provider, law enforcement, or a lawyer. We also collect data on why users uninstall applications, in order to determine the types of application behaviors that prompt user dissatisfaction (Chapter 7.4). We then present the limitations of our methods (Chapter 7.5) and discuss the implications of our findings (Chapter 7.6).

7.2 Ratings

We asked 3,115 smartphone users to rate how upset they would be if an application performed certain actions on their phones without user approval. The purpose of the large-scale rating survey was to create an index that ranks the risks of allowing applications to access smartphone resources by degree of user concern.

7.2.1 Methodology

We surveyed Mechanical Turk users with a survey instrument that we pre-tested with a focus group.

Instrument Design and Validation

Our instrument was designed to elicit user concerns about different resources. We faced two design constraints. First, we aimed to measure opinions about risks rather than application features. (For example, a user might view an application that deletes files as useful or harmful depending on whether the deletion was intentional.) Second, we did not want to scare participants by mentioning malware or viruses. We suspected that participants would report high levels of concern for *any* action that they were told is associated with malware. As such, we needed to ensure that respondents were aware that we were asking about undesirable actions, without mentioning how or why those actions were initiated (e.g., by malware).

We performed two preliminary surveys that asked respondents about situations in which applications performed an action "without my knowledge" or "when you believed [the app] had no reason to do so." The results of these surveys were inconsistent. Subsequent interviews revealed that participants were unsure whether the listed actions were negative side-effects or positive features. We conducted one-on-one interviews and a focus group with Craigslist-recruited smartphone users to generate new wording.

We validated our final instrument by asking four smartphone users to take the survey and speak with an interviewer. These participants were selected from applicants on Craigslist to represent a diverse cross-section of smartphone users. We found that the respondents understood that the questions asked about risks rather than features, and they used the full range of the scale. When asked to describe how the scenarios in the questions could occur, all four participants listed both buggy and compromised applications. Some participants also mentioned viruses, bad UI design, or aggressive marketing. This indicated that all four participants had a firm grasp of the meaning of the questions without specifically focusing on malware.

Instrument

The survey began by asking respondents to think about negative side-effects of applications. First, we asked participants to answer a free-response question about applications that they disliked. Next, we prepared respondents for the risk-based questions:

> *Every once in a while, an app might do something on your phone without asking you first. Depending on what the app does to your phone, your feelings could range from indifference (you don't care) to being very upset.*

We then asked participants about various risks:

> *How would you feel if an app [insert risk], without asking you first?*

For example: "How would you feel if an app added new contacts, without asking you first?" Respondents answered using a horizontal five-point scale that ranged from "Indifferent" to "Very upset," with unlabeled intermediate points.

Each survey participant saw 12 questions on one page, selected at random from a set of 99 potential questions. Appendix B shows the risks that participants were asked about. We compiled the set of questions by assigning risks to Android, Windows Phone 7, and iOS permissions. The three platforms define a total of 191 permissions, but we grouped equivalent permissions (e.g., "power device on or off" and "force device reboot") and discarded irrelevant permissions (e.g., "enable application debugging") to arrive at 54 permissions. We assigned risks to the permissions using documentation and domain expertise. Some permissions are associated with multiple risks, and we assigned least four risks to each type of smartphone data[1]: "publicly shared your [data type]," "shared your [data type] with your friends," "shared your [data type] with advertisers," "sent copies of [data type] to their servers (but didn't share them with anyone else)."

On the last page of the survey, we collected demographic information.

Deployment and Demographics

We deployed the survey on Mechanical Turk for 13 days. Participants were paid $1 each for completing the survey, and we limited the survey to respondents in the United States. We filtered responses for validity based on users' survey completion time, responses to short open-ended questions, and the self-reported type of phone. (We discarded responses with implausibly short times, nonsense answers for the open-ended questions, or non-smartphones.) After filtering the responses, we obtained 3,115 valid responses from smartphone users.

Participants' ages ranged from 18 to 80 (μ=29.7), while 47.9% were female and 51.9% were male. Although the population was younger than the U.S. population overall (65% of respondents

[1]Due to a survey programming error, we accidentally omitted one of these risks for three types of data.

Highest completed level of education	Respondents
Some high school	1.5%
High school diploma or GED	11.2%
Some college or Associate degree	42.3%
Bachelor's degree	28.3%
Some graduate school	5.2%
Master's degree	9.5%
Doctorate or professional graduate degree (Ph.D., J.D., M.D., etc.)	2.0%

Table 7.1. The self-reported education levels of our respondents.

were below the age of 30), it was only slightly younger than U.S. smartphone user demographics [17]. Participants reported many occupations: healthcare workers, software engineers, financial advisors, federal government employees, graphic designers, etc. However, the predominant occupations were students, stay-at-home parents, and the unemployed. Completed levels of education ranged from some high school to doctorates (Table 7.1).

Participants reported owning the following smartphones: 49.5% Android phones, 39.7% iPhones, 7.7% Blackberries, and 1.7% Windows phones. The remainder stated that they owned Palm, Symbian, or multiple phones. There was no incentive to lie about phone ownership because we paid participants regardless of whether they owned smartphones.

7.2.2 Risk Rankings

Our goal is to rank the severity of potential risks based on users' concerns. In our large-scale survey, respondents rated how upset they would be if certain risks occurred. Our resulting metric for the severity of a risk is the percentage of respondents who indicated that they would be "very upset" if the given risk occurred. We refer to this metric as the *VUR rate* (the "very upset" respondent rate). We obtained an average of 376.7 ratings per risk. All but 11 respondents indicated that they would be "very upset" about at least one of the 12 risks.

The highest-ranked risk is "permanently disabled (broke) your phone," with a 98.2% VUR rate. The lowest-ranked risk is "vibrated your phone," with a 15.6% VUR rate. Table 7.2 shows the ten highest-ranked and ten lowest-ranked risks, and Appendix B provides the VUR rates for all of the 99 risks in our survey.

We surveyed respondents about four types of data sharing: public sharing, sharing with friends, sharing with advertisers, and removing the data from the phone without sharing it with another party. Figure 7.1 shows these results for the eleven data types. For all of the data types, publicly sharing the data is approximately twenty percentage points more concerning than sending the data to a server. Sharing with friends and advertisers rank in the middle, between public sharing and sending the data to a server. Notably, illicit location sharing has the lowest or second-lowest VUR rates of the eleven smartphone data types in our survey.

We consider the percentage of "very upset" respondents instead of medians because the re-

Risk	**VUR Rate**
permanently disabled (broke) your phone	98.21%
made phone calls to 1-900 numbers (they cost money)	97.41%
sent premium text messages from your phone (they cost money)	96.39%
deleted all of your contacts	95.89%
used your phone's radio to read your credit card in your wallet	95.15%
publicly shared your text messages	94.48%
deleted all of the information, apps, and settings on your phone	94.39%
publicly shared your e-mails	93.37%
deleted all of your other apps	93.14%
shared your text messages with your friends	92.49%
...	...
inserted extra letters into what you're typing	45.48%
read files that belong to other apps	44.33%
sent your phone's unique ID to their servers (but didn't share it with anyone else)	42.16%
added new browser bookmarks	39.22%
sent the list of apps you have installed to their servers (but didn't share it with anyone else)	34.92%
turned the sound on your phone down really low	36.96%
sent your location to their servers (but didn't share it with anyone else)	29.88%
turned your flash on	29.67%
connected to a Bluetooth device (like a headset)	27.47%
vibrated your phone	15.62%

Table 7.2. The highest- and lowest-ranked risks.

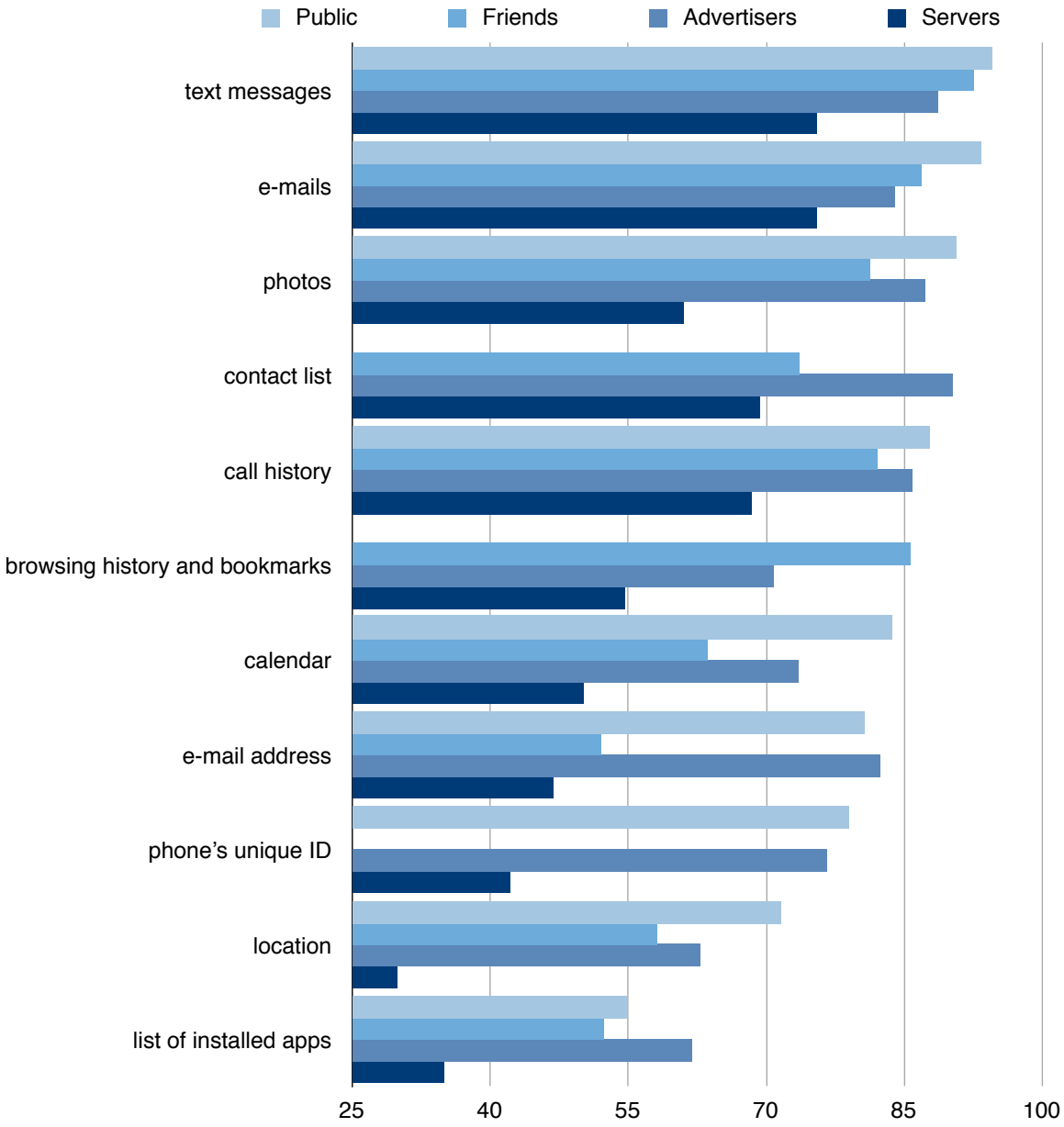


Figure 7.1. The VUR rates for 11 data types. We asked participants to rate how upset they would be if applications illicitly shared their data: "publicly," "with your friends," "with advertisers," and "sent ... to their servers (but didn't share it with anyone else)."

sponses were not normally distributed. (Despite this, ordering the risks by medians returns a very similar ranking.) We observed different amounts of diversity of opinion between risks. Eighteen risks had a standard deviation greater than 1, whereas six had a standard deviation less than .37. In general, the risks with high VUR rates have low standard deviations, whereas the risks with low VUR rates have larger standard deviations. One interpretation is that there is user consensus about what *is* very upsetting, but not about what is *not* very upsetting. It may also be an artifact of our five-point scale: some users might have selected something stronger than "very upset" if such an option were available, which would have resulted in greater variance among high-ranked risks.

Individual respondents' scores are not directly comparable to each other because they received different questions, but we can compare groups of respondents. Women rank risks higher than men do ($\mu_M = 4.47, \mu_W = 4.55$; $p < 0.0005$, $z = -4.269$, Wilcoxon-Mann-Whitney test), although the effect size is very small ($d = 0.18$). People above the age of 50 rank risks higher than people below the age of 30 do ($\mu_{<30} = 4.46, \mu_{>50} = 4.67$; $p < 0.0005$, $z = 5.943$, Wilcoxon-Mann-Whitney test), with a medium effect size ($d = 0.51$). We do not find a significant difference between types of phones ($\chi^2 = 4.487$, $p = 0.6110$, Kruskal-Wallis test).

7.3 Open-Ended Survey

The purpose of the open-ended survey was to validate the large-scale survey ratings with a different metric and associate free-form responses with VUR rates.

7.3.1 Methodology

Instrument

The open-ended survey asked participants the following short essay questions about risks:

1. How would you feel if an app [insert risk], without asking you first?
2. Why would you feel that way?
3. What would you do if this happened?

Each participant was asked about three of nine risks, which we selected based on the results of the large-scale study. We chose the three lowest-ranked risks, three mid-ranked risks, and three of the highest-ranked risks:

- *Lowest-ranked risks:*
  - vibrated your phone
  - connected to a Bluetooth device (like a headset)
  - turned your flash on

- *Mid-ranked risks:*
  - added new contacts
  - took screenshots when you're using other apps
  - un-muted a phone call
- *Highest-ranked risks:*
  - deleted all of your contacts
  - sent premium text messages from your phone (they cost money)
  - made phone calls to 1-900 numbers (they cost money)

We showed each respondent one of the lowest-ranked risks, one of the mid-ranked risks, and one of the highest-ranked risks. We displayed one page for each risk. Participants could not move backwards in the survey to prevent respondents from altering their responses after seeing "scarier" risks. The last page of the survey collected demographic data.

Deployment and Demographics

We deployed the survey on Mechanical Turk for two days. Participants were paid $8 each. The survey was advertised as being worth $3, with an additional $5 reward for complete sentences and correct grammar. We ran the survey until we had 42 valid responses (with a target of 40) from people in the United States. The participants were evenly split by gender, with an average age of 30.3. All of the respondents said that they have used smartphone applications, with an average of 29 applications installed on their phones.

7.3.2 Results

The VUR rate is a relative metric that allows us to compare risks against each other. However, the metric does not provide us with any context for how users interpret "very upset." Our open-ended survey assigns user-supplied meaning to the metric. It also serves as a second measure to evaluate whether there are differences in users' concerns across risks.

	Low-Ranked Risks				**Mid-Ranked Risks**				**High-Ranked Risks**			
	Avg	vibrate	Blue-tooth	flash	**Avg**	added contacts	screen-shots	un-muted	**Avg**	deleted contacts	$ SMS	$ calls
nothing	**21%**	29%	20%	15%	**12%**	18%	0%	15%	**5%**	0%	5%	10%
tinker with app	**33%**	50%	27%	23%	**12%**	6%	33%	0%	**0%**	0%	0%	0%
uninstall the app	**62%**	42%	73%	69%	**74%**	71%	67%	85%	**76%**	67%	80%	80%
contact developer	**12%**	7%	7%	23%	**17%**	18%	8%	23%	**40%**	25%	45%	5%
write a review	**5%**	0%	7%	8%	**14%**	12%	8%	23%	**21%**	25%	20%	20%
contact press	**5%**	0%	13%	0%	**5%**	6%	0%	8%	**5%**	0%	10%	0%
call service provider	**0%**	0%	0%	0%	**2%**	0%	8%	0%	**17%**	17%	15%	20%
replace/wipe phone	**0%**	0%	0%	0%	**5%**	0%	8%	0%	**5%**	8%	0%	10%
contact authorities	**0%**	0%	0%	0%	**2%**	6%	8%	0%	**19%**	25%	15%	20%
pursue legal action	**0%**	0%	0%	0%	**0%**	0%	0%	0%	**12%**	17%	5%	20%

Table 7.3. Forty-two survey respondents told us how they would react if certain risks occurred. We categorized their responses; some responses fall into multiple categories.

We asked participants what they would do if certain risks occurred. Table 7.3 displays the frequency with which participants mentioned certain reactions. Respondents' stated reactions fell in the following categories:

- *Nothing.* Some participants stated that they would ignore the risk or simply reverse the undesirable action.
- *Tinker.* Participants said that they would try to change the application's settings. This was often the first of multiple proposed steps. For example, "I would first try to change the settings so that it doesn't connect. If I can't find the settings to turn such a feature off, I would immediately delete the app."
- *Uninstallation.* The most common recourse was to uninstall the application.
- *Contact the Developer.* Many people said that they would try to contact the developer of the application to complain or request a refund.
- *Reviews.* Some participants said that they would try to make others aware of the application's problems by writing negative reviews. For example, "...for the first time ever, I would probably review [the] app. I would type (probably even in all caps!) about what it does."
- *Contact the Press.* Participants sometimes said that they would warn other users by contacting blogs or "watchdog news groups."
- *Contact the Phone Company.* Several participants said that they would contact their service provider to reverse charges or restore data. Surprisingly, many participants in this category said that they would blame their service providers for negative application behavior. For example, one respondent wrote, "If this happened I would consult my service provider to try and retrieve my contacts, and probably cancel my service." Another said, "I would simply switch to another phone company if I could not uninstall the defective app."
- *Replace or Wipe The Phone.* Although none of the risks in the survey were permanent side-effects, some participants said that they would get a new phone or wipe their existing phone so that it would be like having a new phone. One participant wrote, "[If] this happened and I could not turn off this feature in the settings, then I would not continue using the phone and I would try to either get a refund or to sell it." Another person said that he would *"smash [his] phone to bits."*
- *Contact Authorities.* Some participants said that they would notify authorities about the application's misbehavior so that the application would be punished or removed from the store. For example, "I would call up the FBI or other organizations to look into how my information might have been mishandled."
- *Legal Action.* In some cases, participants wrote that they would seek legal action against the application developer. For example, "...I may seek legal counsel to solve the issue and perhaps receive compensation for the inconvenience and trouble that the application developer put me through."

As Table 7.3 shows, participants' reactions increased in severity from the lowest-ranked risks to the highest-ranked risks. Participants' responses to risks with similar rankings are fairly similar. This supports the validity of the ranking from the large-scale rating survey. For low-ranked risks, participants would attempt to resolve the situation themselves or complain. Responses to mid-range risks contain a greater emphasis on complaining in reviews or to the developer. For high-ranked risks, some would seek help from external parties like service providers or lawyers.

7.4 Reasons for Uninstallation

As part of our large-scale survey, we asked participants to tell us about instances in which they had uninstalled "misbehaving" applications. The purpose of this was to measure the prevalence of risks: a permission system designer might want to guard a low-risk resource with a more severe warning, despite the low VUR rate, if the likelihood of abuse is high.

7.4.1 Methodology

We asked the following question as part of the large-scale survey (described in Chapter 7.2.1):

> *If you have ever un-installed apps because they misbehaved, please tell us what the apps did that you didn't like.*

Respondents answered in short essay format.

7.4.2 Results

Of the 3,115 respondents, 2,427 respondents provided us with short essays about negative experiences with applications that they had uninstalled. We read all of the essay responses and identified 559 responses (17.9% of the total respondents) that described undesirable behaviors that certain to intentional or accidental abuse of resources. We categorized these behaviors into the following categories (Table 7.4):

Spam. Some participants reported having uninstalled applications because the applications caused their phones to send or receive e-mail, text message, or Facebook spam. In order to send spam, applications read users' contact lists and send messages from their accounts. For example, one participant said, "[the app] used information on my contact list to send over a hundred spam texts and emails." Another person wrote that an app "added a contact then sent email from that contact telling my friends I liked the app and they should install it."

Ads in Notifications. Under the rules of the Android Market and iOS App Store, applications are not supposed to use notifications to display advertisements. Despite this restriction, some users

Undesirable behavior	Number of respondents	
Spam	243	(7.8%)
Ads in the notification bar	30	(1.0%)
Other misuses of the notification bar	46	(1.5%)
Drained the battery	85	(2.7%)
Used too much memory	58	(1.9%)
Used too much Internet data	21	(0.7%)
Other negative behaviors	154	(4.9%)

Table 7.4. The number of respondents who report experiencing each of the undesirable application behaviors. Some participants' responses fall into multiple categories. Percentages are from the 3,115 total survey respondents.

report having uninstalled applications for this reason. One participant wrote, "it always put spam in my notification bar, for example: You have won a $50 ATT giftcard! Claim it now!" Our categorization (in Table 7.4) may underestimate the number of people who experienced this; 165 additional participants complained about "pop-up" advertisements, but it was unclear whether they were referring to standard in-application advertisements or abuse of the notification bar.

Other Misuses of the Notification Bar. Other applications misused notifications for reasons other than advertisements. For example, "The only one I remember in particular was one that kept sending me push notifications even though I almost never used it. I know you can turn push off on an iPhone, but I thought it'd be easier to just delete the app." Additionally, a bug in some versions of iOS allows applications to send push notifications even if it has been disabled; some users reported experiencing this, e.g., "Some apps continued to pester me with notifications even though I was more than certain that I had disabled notifications for that app."

Resource Consumption. Some participants felt that applications used up too much battery life, memory, or Internet data. For example, one participant uninstalled several applications because the applications "racked up extra mb of data [while] they were running in the background without me realizing it."

Other Negative Behaviors. Participants also reported a number of behaviors that were too vague or infrequent to categorize. Some vague responses included descriptions of "viruses" and "buggy apps." Infrequent reasons for un-installation included deleting contacts, deleting photographs, recording location, transmitting contacts, and altering contacts.

7.5 Limitations

We discuss the limitations of our methodology to contextualize our discussion of the results.

7.5.1 Demographics

We relied on Mechanical Turk workers for survey data. As discussed in Chapter 7.2.1, the workers who completed our study did not proportionately represent the smartphone population in terms of occupation. Our survey did not reach many highly-paid professionals, who may have different concerns. However, our survey was taken by a large number of participants with varying ages and socio-economic statuses. Secondary studies may be needed to target specific groups that could plausibly have their own privacy and security concerns, such as doctors (patient data), lawyers (client data), or executives (corporate data).

7.5.2 Ranking Questions

The ratings and open-ended questions are not absolute measures of user concern because our surveys explicitly asked respondents about privacy and security. Surveys that directly ask questions about privacy suffer from inflated user concerns about privacy [44] and therefore are not a reliable measure of the absolute level of concern. We expect this applies to our study as well. We intentionally primed respondents to think about the negative side-effects of applications because we did not want users to mix risks and features. Instead, our surveys provide a basis for comparing risks against each other. The same set of priming biases are applied equally to all of the risks presented in the surveys, so this effect should not influence our ranking.

The rating and open-ended questions also rely on self-reported data. Users might act differently when confronted with actual problems on their phones. We do not claim to predict future user behavior. As with the priming bias, we do not believe that self-reporting affects the validity of our ranking because this bias is equally present for all risks.

7.5.3 Uninstallation Question

Participants may have failed to list their negative experiences with applications due to forgetfulness or uncertainty over the open-ended question. Additionally, we only collect information on negative experiences that upset participants enough to uninstall the offending applications; users may have simply ignored other negative experiences that are not described here. On the other hand, we can assume that all of the participants' stories are true because there was no incentive to lie. Consequently, the statistics in Table 7.4 should be viewed as a lower bound: the true rate of negative experiences may be higher.

7.6 Discussion

We discuss the implications of our findings.

7.6.1 Rankings

Causes of Concern. The risks that evinced the highest levels of concern involve permanent data loss or financial loss (e.g., sending premium text messages or spying on credit card numbers). The lowest-ranked risks pertain to phone settings or sending data to servers. Unlike the highest-ranked risks, many of the lowest-ranked risks are revertible: settings can be reset, unwanted browser bookmarks can be removed, the phone's vibrator can be turned off, etc.

Data Sharing. Respondents' concerns about illicit data sharing depend on who the data is being shared with. As Figure 7.1 shows, participants discriminate between publicly sharing data and sharing data with only the application's developers. For some types of data (e.g., contacts, e-mail address), there is a large difference in the VUR rates for sharing with advertisers and other types of sharing. Based on this finding, we suspect that warnings about data access that do not specify where the data is being sent to do not provide users with enough information to gauge the risk of sharing the data with the application. This motivates further work on tools like AppFence [79] that tell users whether data is being sent to advertisers or other known third parties. Alternately, developers could provide annotations that reflect their privacy policies, and this information could be incorporated into warnings or data access requests.

Location. Most mobile privacy and security research has focused on location. However, we find that improper location sharing is not viewed as dangerous in comparison to the other risks of using applications. All of the location-related risks rank in the bottom half of risks, and location is the second-lowest data type. Consequently, we believe that the privacy community should refocus their efforts on other types of smartphone data that evoke higher levels of user concern, such as text messages, photos, and contacts.

Android Warnings. We can compare our ranking to Android's categorization of permissions. Android divides permissions into three levels of severity. We find that their categorization differs from our survey respondents' concerns in many cases. For example, Android places the `SET_TIME` permission in the highest-severity category, yet it falls in the bottom third in our ranking. As another example, access to photos ranks in the top quartile in our study, yet Android does not restrict access to photos with any permission at all.

iOS Warnings. We can also compare our ranking to iOS's selection of warnings. iOS only prompts users for consent for location data and pop-up notifications. However, we find that both rank low in comparison to other privileges; this may indicate that iOS does not ask users about the correct privileges. Although iOS applications cannot perform all of the actions in our ranking, they can perform many of the actions that rank higher than location and notifications without a

consent dialog. Given our ranking, iPhone users' complaints about the lack of a consent dialog for contacts [42] is not surprising.

Service Providers. In our qualitative study, 19% of participants said that they expect their service provider to remedy any data loss or data theft. Several participants stated that they would consider switching service providers if an application severely misbehaved. In practice, service providers do not provide backup services by default, nor do they control what applications are listed in application markets. This suggests that some users may not understand the security or liability implications of installing and using smartphone applications. In contrast, some participants said that they would contact their service providers to refund fraudulent SMS or phone charges; this expectation is likely well-founded for some service providers.

7.6.2 Uninstallation

Nearly 8% of all participants say that they have uninstalled applications because of spam. Spam is related to several high-VUR risks: sending text messages (90.42%), spamming contacts with event invitations (89.73%), sending spam to the user's contact list (88.63%), and sending spam from the user's e-mail account (82.76%). Consequently, emphasizing the significance of permissions that are associated with high-VUR risks would simultaneously promote spam prevention.

Fewer participants reported dissatisfaction due to notification abuse or resource consumption. The risks related to notifications and resource consumption are low-VUR risks: notifications rank 83^{rd} (51.90%) and draining the battery ranks 75^{th} (55.61%). Although these risks might deserve slightly more scrutiny than the other low-VUR risks, our study finds that fewer than 3% of respondents mentioned these risks as causes of uninstallation. However, further research is needed to confirm the true rate of dissatisfaction because our study represents a lower bound.

Among the behaviors that we classified as "other negative behaviors," most correspond to mid- or high-VUR risks. However, none of these behaviors were individually frequent enough to suggest that they might be significant enough to require additional emphasis in a permission system.

7.7 Conclusion

We surveyed 3,115 smartphone users on Mechanical Turk about potential risks of smartphone applications. Participants rated how upset they would be if the risks occurred. From this data, we developed a ranking of risks by user concern. A follow-up, open-ended survey of 41 smartphone users found that users view the lowest-ranked risks are annoyances that they can resolve, whereas the highest-ranked risks are serious offenses that may require external parties. Our ranking could be used to guide warning design, and our results show that location is not a high-ranked user concern. We also found that some users hold service providers responsible for abusive applications.

7.8 Acknowledgments

We thank Serge Egelman, Coye Cheshire, and Galen Panger for their insightful comments regarding survey design and data presentation.

Chapter 8

How To Ask For Permission

8.1 Introduction

Currently, there is no consensus on the best way to communicate permission information to users. Chapter 6 demonstrates that Android's current permission system does not adequately inform or engage most users. iOS only asks users to consent to two permissions; the remainder of the permission system is opaque to users. A number of research proposals argue for the use of trusted UI in permission systems [80, 85, 125, 158], but these have seen minimal adoption in the real world because they do not scale to an entire permission system.

We advocate for a new approach to permission systems: choosing the most appropriate permission-granting mechanism independently for each permission. Our approach takes the unique requirements and constraints of each permission into account. This is a departure from existing platforms, which have applied a single permission-granting mechanism to all permissions. We provide a set of guidelines for selecting from among permission-granting mechanisms.

In this chapter, we enumerate the permission-granting mechanisms that are available to platform designers and discuss their strengths and weaknesses. We take into account a set of usability principles from past literature and rank the permission-granting mechanisms according to the quality of their user experiences. Based on this, we develop a preliminary decision procedure for choosing the most appropriate permission-granting mechanism for each permission.

We apply our decision procedure to a set of smartphone application permissions. We find that the permission-granting mechanisms that afford the best user experiences can be applied to a majority of the permissions. In order to facilitate future work, we built prototypes of permission-granting mechanisms for several permissions. Our prototypes incorporate initial feedback from five iOS users. Future work is needed to evaluate the effectiveness of our guidelines and the usability of new permission-granting mechanisms.

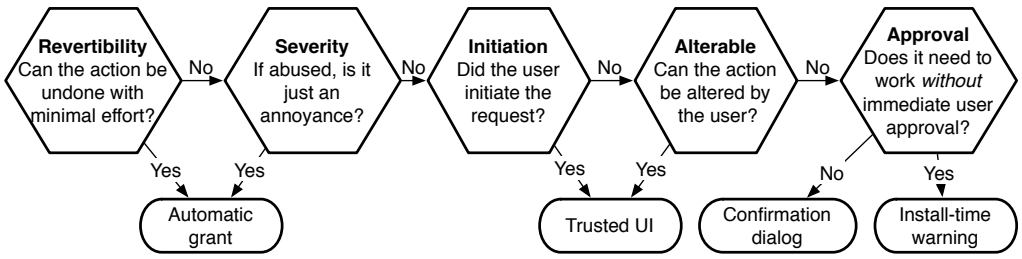


Figure 8.1. A guide to selecting between the different permission-granting mechanisms.

8.2 Guiding Principles

Permission-granting UI elements usually involve closed-form questions: the user is given a choice between granting or denying the permission. Krosnick and Alwin's dual path model [40, 95] is a standard model used to understand human behavior when faced with closed-form questions. According to this model, humans rely upon two distinct strategies. A user who is *optimizing* reads and interprets the question, retrieves relevant information from long-term memory, and makes a decision based on her disposition and beliefs. In contrast, a user who is *satisficing* does not fully understand the question, retrieves only salient cues from short-term memory, and relies on simple heuristics to make a decision. Based on this model, we discuss two principles guiding our design.

8.2.1 Conserve User Attention

Human attention is a shared, finite resource [40]. Its use should be infrequent, and uncontrolled use can lead to a "tragedy of the commons" scenario. Repetition causes *habituation*: once a user clicks through the same warning often enough, he or she will switch from reading the warning before clicking on it to quickly clicking through it without reading (i.e., satisficing). The link between repetition and satisficing has been observed by other researchers for Windows UAC dialogs and browser security warnings [57, 111, 141]. We similarly observed evidence of habituation leading to satisficing in Android (Chapter 6). Users pay less attention to each subsequent warning, and this effect carries over to other, similar warnings [41].

Principle 1: Conserve user attention; use it only for permissions that have severe consequences.

8.2.2 Avoid Interruptions

Users generally do not set out to complete security-related tasks. Instead, they encounter security information when they are trying to check their e-mail, surf the web, or perform some other typical task. Security dialogs often interrupt these primary tasks; for example, Android install-time warnings stand between a user and his primary goal of installing the application.

Users have low motivation to consider interruptive security mechanisms. Instead, they want to return to their primary tasks. Users will click through dialog boxes that interrupt their primary tasks without fully understanding or evaluating the consequences. In other words, interruptions lead to satisficing. Good et al. found that users dismiss or ignore interruptive warnings [75]. In a study of Windows UAC consent dialogs, Motiee et al. observed that most study participants clicked through illegitimate UAC prompts while completing tasks [111].

Principle 2: Avoid interrupting the user's primary task to ask the user to make a security decision.

8.3 Permission-Granting Mechanisms

Permission systems can grant permissions to applications using four basic mechanisms: automatic granting, trusted UI, runtime consent dialogs, and install-time warnings. We discuss these permission-granting mechanisms in order of preference, based on the amount of user attention that they consume. Figure 8.1 summarizes our proposed selection criteria.

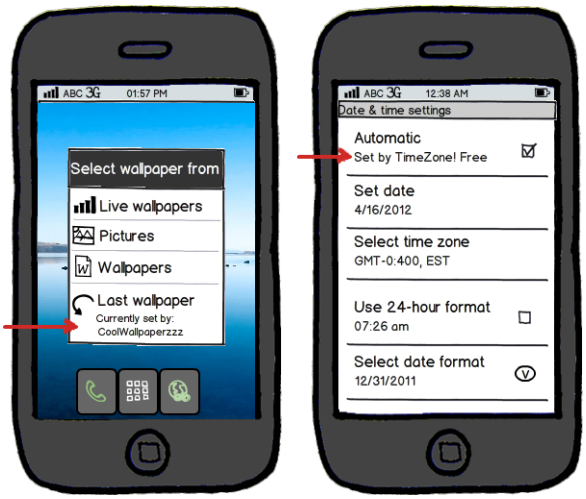
8.3.1 Automatic Grant

Definition. An automatically-granted permission must be requested by the developer, but it is granted without user involvement. We propose that permissions should be automatically granted if they protect easily-revertible or low-severity permissions. For example, changes to the global audio settings are easily revertible, and vibrating the phone is merely annoying. Currently, any web site can play audio or generate pop-up alerts without requesting permission from the user; although this can lead to annoyance, the web is still usable. Web users simply exit web sites that have unwanted music or alerts. Users are aware that they can and should uninstall annoying applications: in Chapter 7, many study participants reported having uninstalled misbehaving applications.

Automatically granted permissions should also include auditing mechanisms to help users identify the source of an annoyance. Auditing can take many forms, such as notifications that attribute actions to applications or "undo" options. These conceptual auditing mechanisms allow users to identify and uninstall abusive applications. Figure 8.2 provides examples of auditing mechanisms for automatically-granted permissions.

Pros. Automatic grants do not require user attention, while still empowering the user to undo the action or uninstall abusive applications.

Cons. Automatically-granted permissions could lead to severe user frustration if a large number of applications were to abuse them. Some permissions may be more attractive as targets for abuse than others. If there is widespread motivation to abuse a permission, then automatic granting may not be appropriate for the permission. The platform can provide deterrents to prevent misuse. For example, developers on platforms with centralized markets are unlikely to abuse permissions in the presence of an auditing mechanism because they do not want users to write negative reviews.



(a) The user can see which application is responsible for the current wallpaper and revert to the last one.

(b) The user can see which application last set the time.

Figure 8.2. Examples of auditing for automatic grants.

Selection Criteria. A permission should be automatically granted if it is easily revertible or low severity. Severity should be determined by surveying users.

8.3.2 Trusted UI

Definition. Trusted UI elements appear as part of an application's workflow, but clicking on them imbues the application with a new permission. To ensure that applications cannot trick users, trusted UI elements can be controlled only by the platform. Trusted UI has been incarnated in many forms, including CapDesk [148], access control gadgets [125], and sensor-access widgets [80]. Examples of trusted UI elements include the following:

- *Choosers* let users select a subset of photos, contacts, songs, etc. Choosers can be adapted to serve as permission-granting mechanisms. Figure 8.3 shows an example photo chooser as trusted UI. Web browsers and Windows 8 Metro currently rely on choosers for providing file access. Other existing choosers (e.g., the iOS photo chooser) could be adapted to serve this purpose, for example with the addition of a "select all" option.

- *Review screens* allow users to review, accept, or modify proposed changes. For example, before adding events to a calendar, a conceptual review screen allows the user to see and optionally edit the events (Figure 8.4(a)). iOS relies on review screens for text messaging: after an application composes a message and selects a recipient, the OS shows the user an

(a) The application wants to read the user's photos, so it opens this chooser.

(b) After clicking on the right arrow, the user selects two albums.

Figure 8.3. A photo chooser. The entire screen is trusted UI; after the application selects the appropriate photos, the chooser would return the photo files to the application.

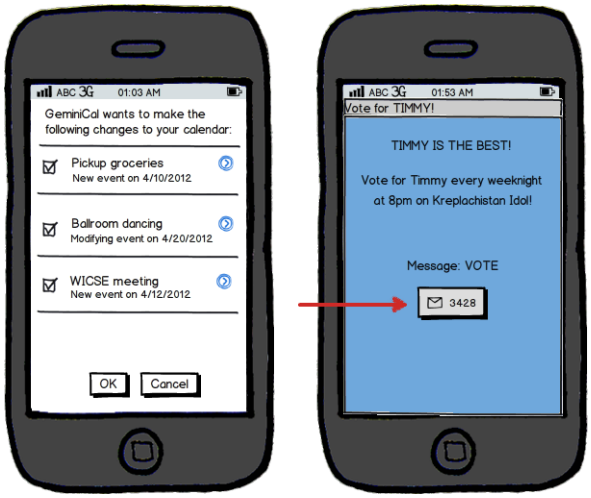
editable preview of the text message. The message is sent only after the user clicks the send button.

- *Embedded buttons* allow a user to grant permissions as part of the natural flow. For example, a user who is sending an SMS message from a third-party application will ultimately need to press a button; using trusted UI means the platform provides the button (Figure 8.4(b)). In a trusted UI design, the applications embed placeholder elements, and the platform renders the button over the placeholder at runtime, optionally incorporating parameters such as the recipient of a phone call or SMS.

Strengths. Trusted UI elements are non-interruptive because they are part of the user's primary task, which means users are motivated to interact with them. Roesner et al. showed that interactions with trusted UI matched users' expectations about privacy and security [125]. Motivation and positive expectations lead to optimizing instead of satisficing.

Weaknesses. Not all permissions can be granted through trusted UI elements. In particular, actions that are not user initiated cannot be represented with trusted UI. For example, a security application may prompt the user when it detects malware. Trusted UI can't accommodate this use case because the application performs the action in response to external state, rather than initiated by the user.[1]

[1]Roesner et al. proposed the creation of embedded buttons with additional text such as "permanent access" [125]. We do not view these as trusted UI because they are displayed prior to the desired functionality and are not part of the normal workflow of an application; they are therefore interruptive.



(a) An item review screen for writing to the calendar.

(b) A permission-granting button for sending SMS.

Figure 8.4. Item review screen and permission-granting button.

Trusted UI elements require effort to design because they need to fit applications' workflows and accommodate as much functionality as possible. They also constrain the appearance of applications, so their design needs to be neutral enough to fit most applications. To help trusted UI blend into applications, the platform could allow some degree of customization or allow developers to choose between multiple designs. For example, developers could choose the size, placement, or color scheme of an element. Ideally, their design would involve input from application developers.

Selection Criteria. A permission should be granted with trusted UI if its use is typically user initiated or the action can be altered by the user (e.g., selecting a subset of photos instead of all photos, or modifying the calendar events that the application is adding).

8.3.3 Runtime Consent Dialogs

Definition. Runtime consent dialogs interrupt the user's flow by prompting them to allow or deny a permission. Runtime consent dialogs sometimes contain descriptions of the risk or an option to remember the decision. Windows UAC prompts and the iOS location and notification dialogs are examples of runtime consent dialogs. A platform can use a standardized consent dialog for all relevant permissions (e.g., Figure 8.5(a)), or it can include customized dialogs for different actions (e.g., Figure 8.5(b)).

Strengths. Runtime consent dialogs can be applied to nearly all permissions by changing the text of a standardized consent dialog or creating new customized dialogs.

(a) The iOS location dialog, which is similar to the iOS notification dialog.

(b) A dialog that is specific to Bluetooth pairing.

Figure 8.5. Runtime consent dialogs.

Weaknesses. As mentioned in Chapter 8.2, runtime consent dialogs encourage satisficing because they are interruptive, repetitive, and overused.

From an application functionality standpoint, runtime consent dialogs are not well suited to actions that need to be performed without the user's immediate consent. For example, a security application might delete all of a user's text messages if the phone is stolen; in this scenario, immediate user approval is not possible because the user does not have physical control of the phone at the time the action needs to occur.

Selection Criteria. Runtime consent dialogs should be used for permissions that cannot be automatically granted or represented with trusted UI. Runtime consent dialogs should not be used if the permission needs to be used in the future without immediate user consent.

8.3.4 Install-Time Warnings

Definition. Install-time permission warnings integrate permission granting into the installation flow. Installation screens list the application's requested permissions. In some platforms (e.g., Facebook), the user can reject some install-time permissions. In other platforms (e.g., Android and Windows 8 Metro), the user must approve all requested permissions or abort installation.

Strengths. Install-time warnings can be applied to any type of permission. Unlike runtime consent dialogs, install-time warnings support situations in which an application needs advance approval of a restricted action. They are also easy to implement.

Weaknesses. Install-time warnings are interruptive because they hinder the user's primary goal of installation. They are also repetitive: users see nearly-identical install-time warnings every time they install an Android application, which results in satisficing (Chapter 6). Compounding their monotony, install-time warnings also suffer from being similar to EULAs. Users typically ignore EULAs [75], and their habituation to EULAs extends to other types of indicators that resemble EULAs [41]. This implies that install-time warnings may not adequately capture users' attention.

In Chapter 6, we also found that users may incorrectly believe that an application cannot use a permission granted during installation until the user confirms its usage at runtime. Several study participants felt that applications could not send text messages without runtime confirmation, regardless of the application's permissions. This indicates that install-time permissions do not match user expectations about when permission will be granted. Another experiment by Roesner et al. similarly found that users do not expect install-time warnings to grant a permission without additional confirmation [125].

Selection Criteria. Use install-time warnings only if no other permission-granting mechanism is appropriate.

8.3.5 Multiple Mechanisms

A platform designer might be tempted to provide multiple mechanisms for the same permission. For example, Android developers have two options for sending SMS messages: (1) they can use trusted UI, or (2) they can forgo trusted UI by using a permission with an install-time warning. In practice, developers often choose the latter for reasons other than functionality [64]. Developers might prefer to design their own UI or be unaware of trusted UI. When developers choose to use an install-time warning or runtime consent prompt instead of trusted UI, the platform designer's intention of conserving user attention is not realized. In general, developers' decisions to use a suboptimal permission-granting mechanism will negatively impact the overall platform.

Platform designers should not enable multiple mechanisms for the same permission without carefully considering developer incentives. Without proper incentives, developers may select the mechanism that is less favorable from the user's and/or platform designer's perspective, negating any benefits of the alternative mechanism.

Selection Criteria. Multiple mechanisms should be avoided. When they are unavoidable, the platform should disincentivize the use of the less-preferable mechanism.

8.4 Expert Review

Some platforms subject some or all applications to an expert security and privacy review. A review process can be used to verify that automatically granted permissions are not abused, thereby increasing the number of automatically granted permissions and reducing the burden that is placed on users' attention. For example, Apple automatically grants all permissions except for location

and notification; developers coarsely specify the privileges they need in their `Info.plist` files. Alternately, a review process can ensure that applications are using the most dangerous permissions appropriately. For example, Google reviews extensions with the most dangerous permissions (i.e., full local machine access). If a platform includes multiple mechanisms, a review process could be used to discourage developers from using the less-desirable mechanism.

We recommend against relying on reviews *in lieu of* user interaction. Experts cannot predict individual privacy decisions. For example, consider an application that periodically collects and transmits the user's running applications: some users may approve this, while others may not want the application to receive this information so frequently. Apple has approved applications that violate user expectations, leading to user outcry [42]. Instead, platforms should use reviews in a complementary manner. Apple could use reviews to ensure that automatically-granted permissions are not abused but still provide users with auditing mechanisms (e.g., attributions and notifications).

8.5 Applying Our Guidelines

We systematically assigned permission-granting mechanisms to a set of permissions, according to our guidelines. We then reviewed 20 iPhone applications to evaluate how the user experience would change in our proposed permission granting system. Last, we prototyped 8 of the permissions that use auditing or trusted UI.

8.5.1 Permission Assignment

To perform the assignment, we first created a set of platform-neutral operating system permissions by combining Android permissions, Windows Phone 7 capabilities, iOS prompts, and the Mozilla WebAPI. (Our guidelines are not platform- or smartphone-specific.) After grouping redundant permissions and dividing broad permissions,[2] we arrived at 75 platform-neutral permissions. We then considered each permission individually. We devised auditing and trusted UI mechanisms as appropriate. We used the set of user concerns from Chapter 7 to identify low-severity permissions.

We find that runtime consent dialogs and install-time warnings cannot be avoided (given that we are unwilling to disable application features), but only a minority of permissions require runtime consent dialogs or install-time warnings. Based on our analysis, 60% of the permissions can be automatically granted, 21% can be represented with trusted UI, 12% require runtime consent dialogs, and 7% would need install-time warnings. Although we could not represent all permissions with automatic grants or trusted UI, our assignment demonstrates that the majority of permissions can be handled with non-interruptive permission-granting mechanisms. Appendix C lists, defines, and categorizes all of the permissions.

[2]For example, we grouped Android's "automatically start at boot" and "make application always run" permissions into a "run all the time" permission, and we divided the "read phone state and identity" permission into "read phone state" and "read phone identity" permissions.

Assigning mechanisms to permissions without compromising application functionality can be complex. In some cases, it may require changes to APIs. For example, we decided to use an embedded button to control the camera permission. After examining applications that take photos, we found that past proposals for controlling camera access with trusted UI [125, 80] would not satisfy many popular applications:

1. A trusted preview screen displays the current camera feed on a portion of the screen, so that users can see what their applications may be recording. However, trusted preview screens cannot be used for applications that omit or partially cover the camera preview as part of their design (e.g., augmented reality).
2. Embedded buttons may not work for applications that capture future photos. Roesner et al. proposed scheduling buttons (i.e., a small trusted calendar), but future events may not conform to predetermined metrics. For example, a bicycling application may automatically take photos at regular distance intervals during a ride. In our design, applications have to use the trusted video button instead; a video recording notification will then flash until the photography is complete.
3. An embedded button is not sufficient when applications apply realtime filters, e.g., to preview a sepia photo. These applications access video data *before* the user presses a button. This can be handled by creating an optional trusted preview screen that renders effects on behalf of applications. (This can be implemented with existing technology, such as WebGL shaders [102].)

Similarly, other permissions might also require changes to APIs to support diverse use cases. For example, having separate API calls for Bluetooth, WiFi, and other settings instead of a single settings API allows each one to have its own permission-granting mechanism.

8.5.2 Preliminary Evaluation of Impact

We reviewed the twenty most popular free iPhone applications to evaluate how our proposed permission system would impact users' and developers' experiences. We manually tested the applications to match their behavior to the 83 API calls discussed in Chapter 8.5. Our results suggest that our approach could potentially provide a feasible basis for a permission system.

Users. iOS is the only smartphone platform that currently uses runtime consent dialogs, and our evaluation shows that our proposal would not degrade the iOS user experience. A user would see an average of 0.25 runtime consent dialogs per application ($min = 0, max = 1$) under our proposed permission system. All of these runtime consent dialogs already exist in iOS. Our proposal would actually decrease the number of runtime consent dialogs because we automatically grant access to the notification API instead of prompting users as iOS does.

Developers. We find that most applications would not require changes to use trusted UI elements because they already use them (e.g., photo choosers). The average application would contain 0.60 trusted UI elements ($min = 0, max = 3$). Three similar applications would need changes: they

would have to replace their camera buttons with the trusted camera button, replace their custom contacts choosers with the trusted contacts chooser, and give their preview filters to a trusted preview window. Apart from trusted UI, other changes to permissions would not impact developers.

8.5.3 Prototype

We modified Android 4.0.4 to include new permission-granting mechanisms for eight permissions:

- vibrating the phone (notification for auditing)
- turning the camera flash on or off (notification for auditing)
- setting the time (an annotation in Date & Time Settings for auditing)
- setting the wallpaper (an annotation in Wallpaper Settings for auditing)
- reading the photo library (a trusted chooser)
- reading the contacts library (a trusted chooser)
- sending SMS messages (an embedded button)
- taking a photo (an embedded button[3])

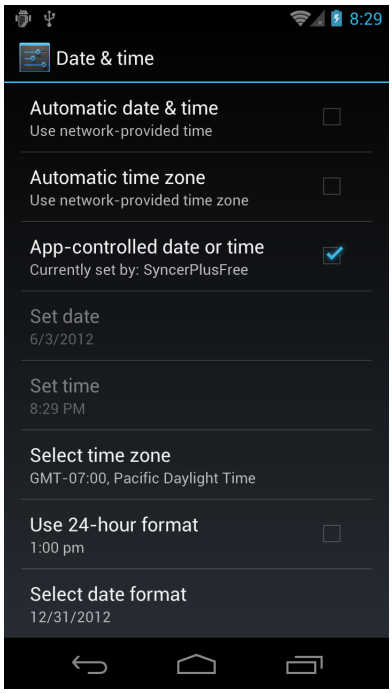
Screenshots of these mechanisms are shown in Figure 8.5.3. We selected Android for the prototype because it is a mature, open source platform; the same mechanisms could be implemented for Boot2Gecko, iOS, or a browser.

We ran a small, qualitative user study to evaluate whether these new mechanisms have the potential to be usable. We recruited five iOS users from Craiglist to test our prototype: three women and two men between the ages of 19 and 51. None of the participants had experience as programmers or graphic designers. The interviews took place in a cafe, and participants were unaware of the security focus. We told participants that we were testing new "smartphone features." None of the participants were familiar with Android, so none had seen the original versions of these screens; consequently, they could not identify the specific alterations that we had made. Each session took thirty minutes, and participants were paid $30 each.

We showed each participant the eight permission-granting mechanisms and asked them to explain the mechanisms to us. For every text label, we asked, "What does that mean?" For every button, we asked, "What do you think would happen if you pressed that button?" Participants were initially asked to not touch the screen. If a participant could not guess the function of a button, we allowed the participant to test it and then asked a follow-up question: "What do you think that button does?" We asked the same questions about other Android UI elements that we had not altered; this served as a control and prevented the participants from fixating on security indicators.

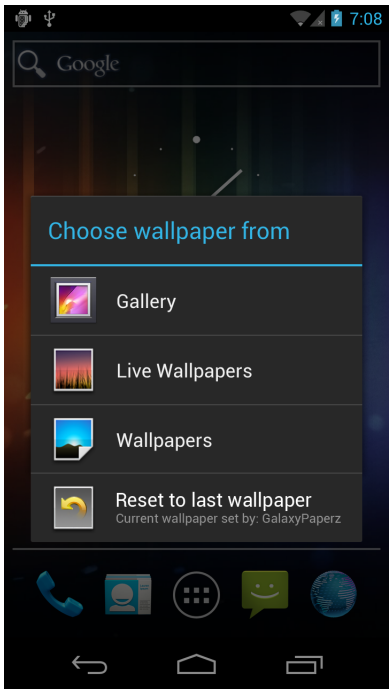
Four of the five participants demonstrated comprehension of all of the mechanisms, with limited confusion over specific wording. For example, our initial wording for the camera flash notification said, "[app name] has turned your flash on"; participants were unsure whether this referred to the camera flash or Adobe Flash, but they were aware that the notification was telling them about

[3]We have not implemented a trusted preview screen that can perform transformations on behalf of applications, which would need to be part of a final implementation, as discussed in Chapter 8.5.1.

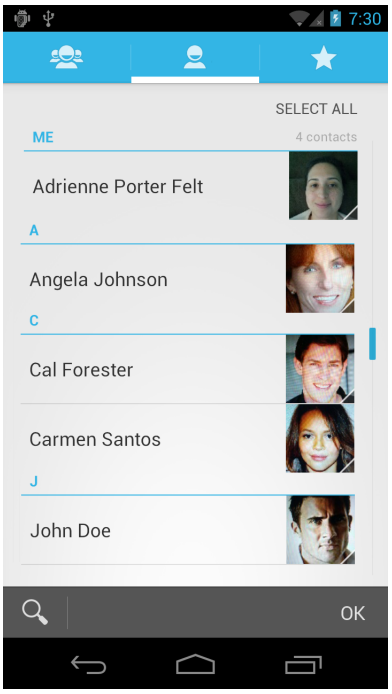


(a) Setting the time: annotation and undo mechanism.

(b) Camera flash: notification.

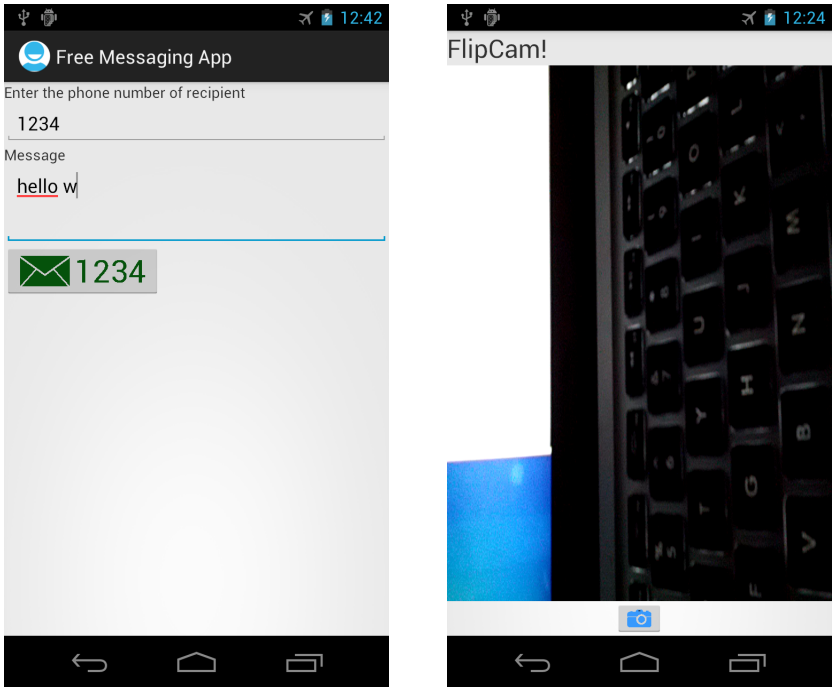


(c) Setting the wallpaper: annotation and undo mechanism.



(d) Reading the contacts library: the trusted contacts chooser.

Figure 8.6. Prototype auditing and trusted UI mechanisms for Android.



(a) Sending SMS: a trusted button that reflects the target number, embedded in a free example application [99].

(b) Taking photos: a trusted button that controls the camera, embedded in a free example application [71].

Figure 8.7. Prototype trusted UI buttons for Android, embedded in applications.

an action that an application had taken. As another example, three participants were initially unsure whether they could select multiple contacts or photos but resolved their uncertainty when they were allowed to touch the screen. These results indicate that notifications, annotations, trusted UI, and embedded buttons may be practical forms of expressing permissions to users, although additional studies will be needed to identify optimal phrases and icons.

One participant, a 39-year-old woman, did not understand any of the prototype mechanisms. However, she also did not understand the original Android UI elements that we asked her about. (E.g., she could not define "automatic date & time: use network-provided time.") This participant expressed general discomfort with her phone, despite having owned an iPhone for two years. She does not use many applications and delegates "difficult" tasks on her phone to family members. Her confusion indicates that additional steps may be needed to reach low-comfort users.

8.6 Future Work

Most current platforms use the same permission-granting mechanism for all permissions. In this chapter, we argue for a new model in which developers choose the most appropriate mechanism for each given permission, with the goal of minimizing the tax on user attention. While our model seems like a plausible direction, further research is needed to perform a comprehensive usability evaluation of a platform that follows our model.

Attention Cost. Although prior research has established that users become habituated to frequent security dialogs, the rate at which users become habituated is not known. If the cost of each interruptive dialog were known, it could be used as a parameter during the design of a permission system: the cost could be used to determine whether the permission system has too many runtime consent dialogs or install-time warnings. It is possible that users experience different habituation rates for runtime consent dialogs and install-time warnings, so separate studies would be needed to establish their respective attention costs.

Display Rates. In order to gauge whether users see interruptive dialogs too frequently, we need to know how often users would see the dialogs in a deployed system. To perform this evaluation without actually deploying the system, a future study should obtain a set of traces from users' phones. This would require long-term monitoring of API calls on participants' phones. The traces would reflect both how users use applications and how applications use resources.

Choosers. We believe that users will optimize when using choosers because choosers are part of the application workflow. However, this hypothesis has not been tested. Future research should test whether users consider their options when selecting items from choosers (i.e., optimize) or click through the default selection (i.e., satisfice).

Embedded Buttons. Although embedded buttons have been proposed by several prior researchers, no one has performed a comprehensive study on the impact that embedded buttons have on application development. Do the buttons restrict functionality? Does their constrained styling negatively

affect applications' user interfaces? Such a study would need to involve input from developers and a large-scale analysis of applications that would need to use embedded buttons.

8.7 Acknowledgements

We thank Matthew Finifter, Serge Egelman, and Devdatta Akhawe for collaborating on the assignment of permissions to permission-granting mechanisms.

Chapter 9

Conclusion

This dissertation demonstrates that modern permission systems strongly benefit platform security. We performed case studies on the Google Chrome extension and Android application permission systems. Based on our findings, future platforms should continue including permission systems, even though they slightly increase the workload of developers.

We considered whether up-front permission declarations protect users from vulnerabilities in benign-but-buggy applications. We found that developers are willing to request close-to-minimal sets of permissions, which reduces the scope of vulnerabilities. Among a set of extensions with real-world vulnerabilities, permissions prevent more than half of the vulnerable extensions from exposing sensitive data (e.g., financial information). Platforms should therefore adopt up-front permission declarations, regardless of when, whether, or how permissions are displayed to users.

We found that Android install-time permission warnings are not usable security indicators for the majority of users. However, some people have altered their behavior based on permissions, which shows that users can be receptive to permission indicators. Future platforms should involve users in the permission-granting process, but they should learn from the flaws in Android's install-time warnings. We argue that permission system designers should avoid taxing user attention with interruptive security dialogs except when it cannot be avoided. Instead, permission systems should employ a diverse set of auditing and trusted UI mechanisms, customized to specific permissions.

We propose a set of guidelines for assigning permission-granting mechanisms to permissions, based on the nature of the mechanisms and permissions. Our initial evaluation shows that this is a promising direction for future research. Future researchers have an opportunity to quantify the limits of user attention and measure the rate at which user attention would be consumed. Several different types of permission-granting and auditing mechanisms have not yet been thoroughly evaluated by usability researchers: ambient notifications, undo mechanisms, and choosers. We believe that additional effort in this research area could yield a permission system that informs and empowers the majority of smartphone users.

Appendix A

Lists of Applications

A.1 Extension Overprivilege

In Chapter 4, we reviewed the following Google Chrome extensions for overprivilege:

Orkut Chrome Extension, Google Similar Pages beta (by Google), Proxy Switchy!, AutoPager Chrome, Send using Gmail (no button), Blog this! (by Google), Fbsof, Diigo Web Highlighter and Bookmark, Woot!, Pendule, Inline Search & Look Up, YouTube Middle-Click Extension, Send to Google Docs, [Non-English Title], PBTweet+, Search Center, Yahoo Mail Widget for Google Chrome, Google Reader Compact, Chromed Movilnet, Ubuntu light-themes scrollbars, Persian Jalali Calender, Intersect, deviantART Message Notifier, Expand, Castle Age Autoplayer Alpha Patched, Patr Pats Flickr App, Better HN, Mark the visited links, Chrome Realtime Search, Gtalk, SpeedyLinks, Slick RSS, Yahoo Avatar, Demotivation.ru ads remover, Short Youtube, (Non-English Title), PPTSearch Edu Sites, Page2RSS, Good Habits, VeryDou, Wikidot Extender, Close Left, iBood, Facebook Colored, eBay Espana (eBay.es) Busqueda avanzada, Keep Last Two Tabs, Google Transliteration Service, Ohio State University Library Proxy Extension, Add to Google Calendar, Rocky

A.2 Extension Vulnerabilities

We selected 100 extensions from the official Chrome extension directory. We have coded extensions as follows: vulnerable and fixed (†), vulnerable but not fixed (‡), and created by Google (*). We last checked whether extensions are still vulnerable on February 7, 2012.

A.2.1 Most Popular Extensions

The 50 most popular extensions (and versions) that we reviewed are as follows:

AdBlock 2.4.6, FB Photo Zoom 1.1105.7.2, FastestChrome - Browse Faster 4.0.6†, Adblock Plus for Google Chrome? (Beta) 1.1.3†, Google Translate 1.2.3.1*‡, Google Dictionary (by Google) 3.0.0*†, Downloads 1, Turn Off the Lights 2.0.0.7, Google Chrome to Phone Extension 2.3.0*, Firebug Lite for Google Chrome 1.3.2.9761†, Docs PDF/PowerPoint Viewer (by Google) 3.5*, RSS Subscription Extension (by Google) 2.1.3*‡, Webpage Screenshot 5.2†, Mail Checker Plus for Google Mail 1.2.3.3, Awesome Screenshot: Capture & Annotate 3.0.4‡, Google Voice (by Google) 2.2.3.4*†, Speed Dial 2.1‡, Smooth Gestures 0.15.2, Xmarks Bookmark Sync 1.0.14, Send from Gmail (by Google) 1.12*, SocialPlus! 2.5.4‡, FlashBlock 0.9.31, AddThis - Share & Bookmark (new) 2.1†, WOT 1.1, Add to Amazon Wish List 1.0.0.4†, StumbleUpon 3.5.18.1†, Google Calendar Checker (by Google) 1.2.1*, Clip to Evernote 5.0.14.9248, Google Quick Scroll 1.8*, Stylish 0.7, Silver Bird 1.9.7.9†, SmoothScroll 1.0.1, Browser Button for AdBlock 0.0.13, TV 2.0.5, Fast YouTube Search 1.2‡, Slideshow 1.2.9†, bit.ly — a simple URL shortener 1.2.1.9, Web Developer 0.3.1, LastPass 1.73.2, SmileyCentral 1.0.0.3‡, Select To Get Maps 1.1.1‡, TooManyTabs for Chrome 1.6.5, Blog This! (by Google) 0.1.1*, TinEye Reverse Image Search 1.1, ESPN Cricinfo 1.8.3†, MegaUpload DownloadHelper 1.2, Forecastfox 2.0.10‡, PanicButton 0.13.1†, AutoPager Chrome 0.6.2.12, RapidShare DownloadHelper 1.1.1.

A.2.2 Randomly Selected Extensions

The 50 randomly selected extensions (and versions) that we reviewed are as follows:

The Independent 1.7.0.3†, Deposit Files Download Helper 1.2, The Huffington Post 1.0.5‡, Bookmarks Menu 3.4.6, X-notifier (Gmail, Hotmail, Yahoo, AOL ...) 0.8.2‡, SmartVideo For YouTube 0.94, PostRank Extension 0.1.7, Bookmark Sentry 1.6.5†, Print Plus 1.0.5.0‡, 4chan 4chrome 9001.47‡, HootSuite Hootlet 1.5, Cortex 1.8.3, ScribeFire 1.7‡, Chrome Dictionary Lite 0.2.6†, Taberareloo 2.0.17, SEO Status Pagerank/Alexa Toolbar 1.6, ChatVibes Facebook Video Chat! 1.0.7†, PHP Console 2.1.4, Blank Canvas Script Handler 0.0.17‡, Reddit Reveal 0.2, Greplin 1.7.3, DropBox 1.1.5, Speedtest.or.th 1, Happy Status 1.0.1‡, New Tab Favorites 0.1, Ricks Domain Cleaner for Chrome 1.1.1, Fazedr 1.6†, LL Bonus Comics First! 2.2, Better Reddit 0.0.4, (non-English characters) 1, turl.im url shortener 1.1, Wooword Bounce 1.2, ntust Library 0.7, me2Mini 0.0.81‡, Back to Top 1.1, Favstar Tally by @paul_shinn 1.0.0.0, ChronoMovie 0.1.0, AutoPagerize 0.3.1, Rlweb's Bitcoin Generator 0.1, Nooooo button 1‡, The Bass Buttons 1.95, Buttons 1.4, OpenAttribute 0.6†, Nu.nl TV gids 1.1.3‡, Hide Sponsored Links in Gmail? 1.4, Short URL 4, Smart Photo Viewer on Facebook 1.3.0.1‡, Airline Checkin (mobile) 1.2102, Democracy Now! 1.1‡, Coworkr.net Chrome 0.9.

Appendix B

Full Results Of User Concern Survey

In Chapter 7, we collected survey data on user concerns. Table B.1 shows the VUR rates for all of the 99 risks that were in our survey, ordered by rank. We have grouped questions about sharing the same data type into the same row. The tightest confidence interval for a VUR rate is $\pm 1.4\%$ at a 95% confidence level, and the widest confidence interval for a VUR rate is $\pm 5.0\%$ at a 95% confidence level. (The confidence interval depends on the number of ratings for a given risk — the average number of ratings was 376.7 — and how close the VUR rate is to 50%.)

Risk	Very Upset Rate
permanently disabled (broke) your phone	98.21%
made phone calls to 1-900 numbers (they cost money)	97.41%
sent premium text messages from your phone (they cost money)	96.39%
deleted all of your contacts	95.89%
used your phone's radio to read your credit card in your wallet	95.15%
publicly shared your text messages	94.48%
deleted all of the information, apps, and settings on your phone	94.39%
publicly shared your e-mails	93.37%
deleted all of your other apps	93.14%
shared your text messages with friends	92.49%
recorded your credit card # when you entered it into a different app	92.35%
publicly shared your photos	90.60%
changed your keylock/pattern/PIN	90.46%
sent text messages from your phone	90.42%
shared your contact list with advertisers	90.19%
spammed your contacts with event invitations	89.73%
made phone calls	89.62%
shared your text messages with advertisers	88.63%
sent spam to people on your contact list	87.95%
publicly shared your call history	87.77%
shared your photos with advertisers	87.26%
shared your e-mails with your friends	86.87%
shared your call history with advertisers	85.80%
shared your browsing history and bookmarks with friends	85.68%
shared your e-mails with advertisers	83.96%
publicly shared your calendar	83.68%
recorded the passwords that you enter into other apps and websites	83.38%
sent spam from your e-mail account	82.76%
shared your call history with your friends	82.04%
shared your photos with your friends	81.28%
shared your e-mail address with advertisers	82.31%
deleted or changed files used by other apps on your phone	82.14%
inserted spam messages at the end of a text message you sent	81.15%
hung up your phone when you're talking	81.00%
publicly shared your e-mail address	80.70%
publicly shared your phone's unique ID	78.92%
recorded you speaking with your phone's microphone	78.86%
installed other apps onto your phone	78.46%
took a photo with your front-facing camera	77.30%
muted a phone call when you're talking	77.27%
deleted all of the events on your calendar	76.89%
shared your phone's unique ID with advertisers	76.61%
posted to your Facebook wall	76.30%
used your data plan to download data when you were roaming	75.57%
sent your e-mails to their servers (but didn't share them with anyone else)	75.51%
turned your keylock/pattern/PIN off	74.72%
sent your text messages to their servers (but didn't share them with anyone else)	75.48%
deleted other apps' saved passwords	73.96%
shared your contact list with friends	73.59%
took a photo with your rear-facing camera	73.54%
shared your calendar with advertisers	73.47%
publicly shared your location	71.57%
shared your browsing history and bookmarks with advertisers	70.74%
un-muted a phone call	70.73%
took screenshots when you're using other apps	70.23%
deleted all of your browser bookmarks and RSS feeds	69.87%

Risk	Very Upset Rate
added new contacts	69.57%
sent your contact list to their servers (but didn't share it with anyone else)	69.29%
force quit all your other apps	69.29%
sent your call history to their servers (but didn't share it with anyone else)	68.41%
used your data plan to download data	67.83%
turned your Internet connection off while you were using the Internet	64.84%
logged in to your Facebook account	64.34%
prevented other apps from running	63.99%
shared your calendar with your friends	63.59%
shared your location with advertisers	62.80%
shared the list of apps you have installed with advertisers	61.85%
prevented your phone from being backed up to your computer	61.39%
sent your photos to their servers (but didn't share them with anyone else)	60.95%
used your phone's unique ID to track you across apps	60.33%
changed the time on your phone	60.00%
logged in to your saved Google account	59.38%
shared your location with your friends	58.10%
restarted your phone	57.56%
drained your battery	55.61%
publicly shared the list of apps you have installed	54.77%
sent your browsing history and bookmarks to their servers (but didn't share them with anyone else)	54.59%
set alarms on your phone	54.05%
changed your phone's wallpaper	52.51%
shared the list of apps you have installed with your friends	52.36%
turned the sound on your phone up really high	52.12%
shared your e-mail address with your friends	52.00%
showed you lots of pop-up notifications	51.90%
prevented your phone from being backed up to the cloud	50.52%
sent your calendar to their servers (but didn't share it with anyone else)	50.14%
slowed down your phone	50.00%
disconnected you from a Bluetooth device (like a headset) while you were using the Bluetooth device	48.63%
sent your e-mail address to their servers (but didn't share it with anyone else)	46.87%
turned your WiFi back on when you were on a plane	45.52%
inserted extra letters into what you're typing	45.48%
read files that belong to other apps	44.33%
sent your phone's unique ID to their servers (but didn't share it with anyone else)	42.16%
added new browser bookmarks	39.22%
turned the sound on your phone down really low	36.96%
sent the list of apps you have installed to their servers (but didn't share it with anyone else)	34.92%
sent your location to their servers (but didn't share it with anyone else)	29.88%
turned your flash on	29.67%
connected to a Bluetooth device (like a headset)	27.47%
vibrated your phone	15.62%

Table B.1. The number of respondents who indicated they would be "Very upset" if a given risk occurred. We asked 99 questions about risks; each respondent saw 12.

Appendix C

Categorized Permissions

We defined and categorized 81 platform-neutral permissions in Chapter 8. We list those permissions in Table C.1, which continues for multiple pages.

Property	Android permission	Web APIs	Granting mechanism
declare a widget that runs before the PIN is entered	System tools: disable keylock		Automatic
run all the time	System tools: automatically start at boot, System tools: make application always run		Automatic
keep phone awake (drain the battery)	System tools: prevent device from sleeping		Automatic
turn vibrator on/off	Hardware controls: control vibrator	navigator.vibrate	Automatic
turn camera flash on/off	Hardware controls: control flashlight		Automatic
change wallpaper	System tools: set wallpaper	possibly covered by SettingsLock.set; still unspecified in the specs	Automatic
change the time	System tools: set time zone, Default: set time	covered by SettingsLock.set (settings.time.useNetworkTime, settings.time.timezone, settings.time.msSinceEpoch); might also support adjustSystemClock	Automatic
turn sound up or down	Hardware controls: change your audio settings	possibly covered by SettingsLock.set; still unspecified in the specs	Automatic
set alarm	Your personal information: set alarm in alarm clock		Automatic
add words to the dictionary	Your personal information: write to user defined dictionary		Automatic
change the default locale	System tools: change your UI settings	possibly covered by SettingsLock.set; still unspecified in the specs	Automatic
change the font size and font	System tools: change your UI settings	possibly covered by SettingsLock.set; still unspecified in the specs	Automatic
change the ringtone	System tools: change your UI settings	possibly covered by SettingsLock.set; still unspecified in the specs	Automatic
force quit other apps (task management)	System tools: force stop other applications, System tools: kill background processes, Default: enable or disable application components, Default: prevent app switches, Default: monitor and control all application launching		Automatic
control an app's backup to the cloud or a computer	Default: control system backup and restore		Automatic
change sync settings for an app's own account, when not roaming	System tools: write sync settings		Automatic
view network state	Network Communication: View network state	MobileConnection.cardState, MobileConnection.emergencyCallsPossible, MobileConnection.roaming...etc, Connection.bandwidth, Connection.metered	Automatic

Table C.1. Our categorization of permissions.

Property	Android permission	Web APIs	Granting mechanism
view wifi state, not including current SSIDs	Network Communication: View Wi-Fi state	WifiManager.getNetworks, WifiManager.connected, WifiManager.signalStrength, WifiManager.connectTemp (note that all of these require knowing the SSIDs); Connection.bandwidth, Connection.metered	Automatic
data plan access, including both pull and push, when not roaming	Network Communication: Full Internet access, Network Communication: Receive data from Internet, Network Communication: Download files without notification	normal internet connectivity for websites	Automatic
connect or disconnect from bluetooth devices	Network Communication: Create bluetooth connections, System tools: Bluetooth administration	nsIDOMBluetoothAdapter.startDiscovery, nsIDOMBluetoothAdapter.stopDiscovery, ondevicefound etc	Automatic
connect or disconnect from known wifi/cell networks	System tools: Change network connectivity, System tools: Change Wi-Fi state	according to the WebMobileConnection wiki, connecting to/from cell networks will be covered by SettingsLock.set	Automatic
connect or disconnect from new wifi/cell networks	System tools: Change network connectivity, System tools: Change Wi-Fi state, System tools: write access point name settings	according to the WebMobileConnection wiki, connecting to/from cell networks will be covered by SettingsLock.set	Automatic
read system log (note: this may include private information if companies are careless)	Your personal information: read sensitive log data		Automatic
accelerometer			Automatic
read unique ID; does not need to be IMEI	Phone calls: read identity	will be exposed by WebAPI but not currently attached to anything yet	Automatic
see that a call is beginning (number not visible)	Phone calls: read phone state, Phone calls: intercept outgoing calls		Automatic
send call to voicemail			Automatic
modify phone state: disconnect phone calls, mute the phone, unmute the phone	Phone calls: modify phone state	navigator.TelephonyCall.answer, navigator.telephony.TelephonyCall.hangUp, navigator.telephony.startTone, navigator.telephony.stopTone, etc.	Automatic
mount or unmount an external hard drive that is connected to the device	System tools: mount and unmount filesystems		Automatic
move app resources (files etc.) from SD card to internal storage, or vice versa; only grants ability to move own resources	Default: Move application resources		Automatic
Pop-up notifications			Automatic

Property	Android permission	Web APIs	Granting mechanism
add notifications to status bar	Default: send download notifications		Automatic
sync	System tools: read sync settings, System tools: read sync statistics		Automatic
reorganize other processes	Development tools: limit number of running processes, System tools: reorder running applications		Automatic
Your personal information: read user defined dictionary	Your personal information: read user defined dictionary		Automatic
IPC	Development tools: send Linux signals to applications, Network communication: broadcast data messages to applications, System tools: send package removed broadcast, System tools: send sticky broadcasts, Your messages: send SMS-received broadcasts, Your messages: send WAP-PUSH-received broadcasts		Automatic
measure application storage space	System tools: measure application storage space		Automatic
modify global animation speed	System tools: modify global animation speed		Automatic
install DRM content	Default: install DRM content		Automatic
use mock location sources for testing	Your location: mock location sources for testing		Automatic
enable application debugging	Development tools: enable application debugging		Automatic
read battery statistics	Default: modify battery statistics (this really only allows an app to READ battery statistics)		Automatic
delete other applications' caches	Default: delete other applications' caches, System tools: delete all application cache data, Default: access the cache filesystem		Automatic
system debugging information	Your personal information: retrieve system internal state		Automatic
display or hide the full notifications menu	System tools: expand/collapse status bar	possibly covered by SettingsLock.set; still unspecified in the specs	Automatic for foreground
delete applications	Default: delete applications	navigator.mozApps.uninstall	Runtime dialog
install applications	Default: directly install applications	navigator.mozApps.install	Install-time warning
see that a call is beginning, including the number of the call			Install-time warning
change the number of an outgoing call	Phone calls: intercept outgoing calls		Install-time warning
keylogging other apps (used by input methods)	Default: read frame buffer, Default: record what you type and actions you take, Default: bind to an an input method		Install-time warning
insert keystrokes and keypresses into other applications	Default: press keys and control buttons		Install-time warning

Property	Android permission	Web APIs	Granting mechanism
sync, when roaming	System tools: write sync settings		Runtime dialog
view nearby SSIDs	Network Communication: View Wi-Fi state	WifiManager.getNetworks, WifiManager.connected, WifiManager.signalStrength, WifiManager.connectTemp (note that all of these require knowing the SSIDs); Connection.bandwidth, Connection.metered	Runtime dialog
data plan access, including both pull and push, when roaming	Network Communication: Full Internet access, Network Communication: Receive data from Internet, Network Communication: Download files without notification	normal internet connectivity for websites	Runtime dialog
NFC	Network Communication: Control Near Field Communication	"The NDEF capabilities are, for now, restricted to tag discovery. The tagdiscovered NfcNdefEvent will be fired when a new NDEF tag is discovered."	Runtime dialog
read browser history, bookmarks, and RSS feeds	Your personal information: read browser's history and bookmarks, System tools: read subscribed feeds		Runtime dialog
read location	Your location: coarse (network-based) location, Your location: fine (GPS) location		Runtime dialog
read call history	Phone calls: read phone state, Phone calls: intercept outgoing calls	navigator.telephony.calls, also can put a handler on each incoming call and see who it's with	Runtime dialog
read SMS messages (both sent and received)	Your messages: read SMS or MMS, Your messages: receive MMS, Your messages: receive SMS, Your messages: receive WAP	navigator.sms.getMessage, navigator.sms.getMessages	Runtime dialog
send premium SMS messages	Services that cost you money: send SMS messages	navigator.sms.send	Runtime dialog
turn phone off or reboot phone	Default: power device on or off, Device: force device reboot		Trusted UI
edit bookmarks/RSS feeds	Your personal information: write browser's history and bookmarks, System tools: write subscribed feeds		Trusted UI - Action review
send calendar invitations	Your personal information: send emails to [calendar] guests		Trusted UI - Action review
add, delete, or edit contacts	Your personal information: write contact data	ContactManager.clear, ContactManager.safe, ContactManager.remove	Trusted UI - Action review
view photos		currently assumed that photos will be accessed via the DeviceStorage API, no specific photos API	Trusted UI - Chooser
access audio files that other apps recorded	Default: audio file access		Trusted UI - Chooser

Property	**Android permission**	**Web APIs**	**Granting mechanism**
read contacts	Your personal information: read contact data	ContactsManager.find	Trusted UI - Chooser
read other applications files	Storage: modify/delete USB Storage contents modify/delete SD card contents (READ), Default: access DRM content	DeviceStorage API is one folder for the whole storage system; but also localStorage	Trusted UI - Chooser
write, change, or delete other applications files	Storage: modify/delete USB Storage contents modify/delete SD card contents (WRITE), System tools: format external storage, Default: delete other applications' data	DeviceStorage API is one folder for the whole storage system; but also localStorage	Trusted UI - Chooser
see other accounts (includes e-mail addresses and logins); can be useful if you use other services for login/auth	Your accounts: discover known accounts, Your accounts: view configured accounts, Default: discover known accounts		Trusted UI - Chooser
record with the microphone	Hardware controls: record audio	will be part of the VideoConferencing API	Trusted UI - magic button
use the camera (either front or rear)	Hardware controls: take pictures and videos (note that windows phone separates front and rear facing cameras)	will be part of the VideoConferencing API	Trusted UI - Magic button
full screen	Default: disable or modify status bar		Trusted UI - Magic button
send non-premium SMS messages	Services that cost you money: send SMS messages	navigator.sms.send	Trusted UI - Magic button
place phone calls	Services that cost you money: directly call phone numbers	navigator.telephony.dial	Trusted UI - Magic button
write to the calendar	Your personal information: add or modify calendar events		Trusted UI -Action review

Appendix D

Research Ethics and Safety

The studies described within this paper received approval from the Institutional Review Board (IRB) at the University of California, Berkeley, prior to the commencement of the research. The studies were approved as part of "User Preferences for Application Permissions" (2011-04-3106), "The Effect of Android User Interface Elements on User Behavior" (2011-07-3405), "Testing New Browser Permission Dialogs" (2012-03-4121), and "Mobile App Survey" (2011-10-3677).

We obtained consent from all study participants to share their responses anonymously. All Internet survey data was collected anonymously, and the IP address logs were deleted once responses had been de-duplicated. Laboratory study and focus group participants were not interviewed anonymously, but all data was coded so that only the lead researcher could connect names to participant data. We gave laboratory study and focus group participants an opportunity to ask clarifying questions about the study after their interviews.

Bibliography

[1] "Application fundamentals," Android developer documentation. [Online]. Available: http://developer.android.com/guide/components/fundamentals.html

[2] "Broadcast Intent when network state has changed," Stack Overflow. [Online]. Available: http://stackoverflow.com/questions/2676044/broadcast-intent-when-network-state-has-changend

[3] "Capabilities," S60 5th Edition C++ Developer's Library v2.1. [Online]. Available: http://library.developer.nokia.com/index.jsp?topic=/S60_5th_Edition_Cpp_Developers_Library/GUID-6971B0A2-F79B-4E05-8AF3-BB1FC1932A22.html

[4] "Content Security Policy (CSP)," Google Chrome Extension Documentation. [Online]. Available: http://code.google.com/chrome/extensions/trunk/contentSecurityPolicy.html

[5] "Cross-Origin XMLHttpRequest," Google Chrome Extension Documentation. [Online]. Available: http://code.google.com/chrome/extensions/xhr.html

[6] "Intents and Intent Filters," Android developer documentation. [Online]. Available: http://developer.android.com/guide/components/intents-filters.html

[7] "Manifest.permission," Android developer documentation. [Online]. Available: http://developer.android.com/reference/android/Manifest.permission.html

[8] "Npapi plugins," Google Chrome Extension Documentation. [Online]. Available: http://code.google.com/chrome/extensions/npapi.html

[9] "Permissions reference," Facebook Developers. [Online]. Available: https://developers.facebook.com/docs/authentication/permissions/

[10] "SELinux Project Wiki." [Online]. Available: http://selinuxproject.org/page/Main_Page

[11] "Tabs," Google Chrome Extension Documentation. [Online]. Available: http://code.google.com/chrome/extensions/tabs.html

[12] "The Add-on Review Process and You," Mozilla Add-ons Blog. [Online]. Available: http://blog.mozilla.com/addons/2010/02/15/the-add-on-review-process-and-you

[13] "Why is using JavaScript eval function a bad idea?" Stack Overflow. [Online]. Available: http://stackoverflow.com/questions/86513/why-is-using-javascript-eval-function-a-bad-idea

[14] "Trusted computer system evaluation criteria (orange book)," Department of Defense, Tech. Rep. DOD 5200.28-STD, December 1985.

[15] "Device APIs Requirements: W3C Working Group Note 15 October 2009," W3C Working Group Notes, 2009. [Online]. Available: http://www.w3.org/TR/2009/NOTE-dap-api-reqs-20091015/

[16] "How Consumers Interact with Mobile App Advertising," Harris Interactive Survey, December 2011. [Online]. Available: http://www.pontiflex.com/download/harrisinteractive.pdf

[17] "US Smartphone Owners by Age," comScore Data Mine, June 2011. [Online]. Available: http://www.comscoredatamine.com/2011/06/us-smartphone-owners-by-age

[18] M. Ackerman, L. Cranor, and J. Reagle, "Privacy in e-commerce: examining user scenarios and privacy preferences," in *Proceedings of the ACM Conference on Electronic Commerce*, 1999.

[19] L. Adamski, "Security severity ratings," MozillaWiki. [Online]. Available: https://wiki.mozilla.org/Security_Severity_Ratings

[20] B. Adida, A. Barth, and C. Jackson, "Rootkits for JavaScript Environments," in *Proceedings of the Workshop on Web 2.0 Security and Privacy (W2SP)*, 2009.

[21] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe, "Efficient and language-independent mobile programs," in *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, 1996.

[22] AdMob, "AdMob Mobile Metrics Report," 2010. [Online]. Available: http://metrics.admob.com/wp-content/uploads/2010/02/AdMob-Mobile-Metrics-Jan-10.pdf

[23] D. Akhawe, P. Saxena, and D. Song, "Privilege Separation for HTML5 Applications," in *Proceedings of the USENIX Security Symposium*, 2012.

[24] Android Open Source Project, "Android Security Overview," 2012. [Online]. Available: http://source.android.com/tech/security/index.html

[25] D. Anthony, D. Kotz, and T. Henderson, "Privacy in location-aware computing environments," *IEEE Pervasive Computing*, vol. 6, no. 4, 2007.

[26] S. Artzi, M. Ernst, A. Kiezun, C. Pacheco, and J. Perkins, "Finding the needles in the haystack: Generating legal test inputs for object-oriented programs," in *Proceedings of the Workshop on Model-Based Testing and Object-Oriented Systems*, 2006.

[27] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett, "VEX: Vetting Browser Extensions For Security Vulnerabilities," in *Proceedings of the USENIX Security Symposium*, 2010.

[28] L. Barkhuus, "Privacy in location-based services, concern vs. coolness," in *Proceedings of the Workshop on Location System Privacy and Control at MobileHCI*, 2004.

[29] L. Barkhuus and A. Dey, "Location-based services for mobile telephony: a study of users' privacy concerns," in *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction (INTERACT)*, 2003.

[30] D. Barrera, H. Kayacik, P. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to Android," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2010.

[31] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, "Protecting Browsers from Extension Vulnerabilities," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[32] A. Barth, "More secure extensions, by default," Chromium Blog, February 2012. [Online]. Available: http://blog.chromium.org/2012/02/more-secure-extensions-by-default.html

[33] A. Berman, V. Bourassa, and E. Selberg, "TRON: process-specific file protection for the UNIX operating system," in *Proceedings of the USENIX Technical Conference Proceedings*, 1995.

[34] D. J. Bernstein, "The qmail security guarantee." [Online]. Available: http://cr.yp.to/qmail/guarantee.html

[35] A. Besmer and H. R. Lipford, "Users' (mis)conceptions of social applications," in *Proceedings of Graphics Interface*, 2010.

[36] J. R. Bettman, *An Information Processing Theory of Consumer Choice*. Addison-Wesley Publishing Company, 1979.

[37] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: splitting applications into reduced-privilege compartments," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[38] N. J. Blunch, "Position Bias in Multiple-Choice Questions," *Journal of Marketing Research*, 1984.

[39] E. Bodden, A. Sewe, J. Sinschek, and M. Mezini, "Taming reflection: Static analysis in the presence of reflection and custom class loaders," CASED, Tech. Rep. TUD-CS-2010-0066, Mar. 2010. [Online]. Available: http://www.bodden.de/pubs/TUD-CS-2010-0066.pdf

[40] R. Böhme and J. Grossklags, "The Security Cost of Cheap User Interaction," in *Proceedings of the New Security Paradigms Workshop (NSPW)*, 2011.

[41] R. Böhme and S. Köpsell, "Trained to accept? A field experiment on consent dialogs," in *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*, 2010.

[42] C. Bonnington, "Apple Says Grabbing Address Book Data Is an iOS Policy Violation," Wired: Gadget Lab, February 15 2012. [Online]. Available: http://www.wired.com/gadgetlab/2012/02/apple-responds-to-path/

[43] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2002.

[44] A. Braunstein, L. Granka, and J. Staddon, "Indirect Content Privacy Surveys: Measuring Privacy Without Asking About It," in *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2011.

[45] T. Buchanan, C. Paine, A. N. Joinson, and U.-D. Reips, "Development of measures of online privacy concern and protection for use on the Internet," *Journal of the American Society for Information Science and Technology*, 2007.

[46] B. Chess, Y. T. O'Neil, and J. West, "JavaScript Hijacking," Fortify, Tech. Rep., 2007.

[47] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing Inter-Application Communication in Android," in *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2011.

[48] G. Cluley, "Windows Mobile Terdial Trojan makes expensive phone calls," naked security by Sophos. [Online]. Available: http://www.sophos.com/blogs/gc/g/2010/04/10/windows-mobile-terdial-trojan-expensive-phone-calls/

[49] F. Coker, "Writing SE Linux policy HOWTO," 2004. [Online]. Available: http://www.lurking-grue.org/writingselinuxpolicyHOWTO.html

[50] S. Consolvo, I. E. Smith, T. Matthews, A. LaMarca, J. Tabert, and P. Powledge, "Location disclosure to social relations: why, when, & what people want to share," in *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*, 2005.

[51] L. F. Cranor, "A framework for reasoning about the human in the loop," in *Proceedings of the USENIX Conference on Usability, Psychology, and Security*, 2008.

[52] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Software: Practice and Experience*, vol. 34, no. 11, 2004.

[53] B. Darwell, "Facebook platform supports more than 42 million pages and 9 million apps," Inside Facebook, April 2012. [Online]. Available: http://www.insidefacebook.com/2012/04/27/facebook-platform-supports-more-than-42-million-pages-and-9-million-apps/

[54] R. Dhamija, J. D. Tygar, and M. Hearst, "Why phishing works," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2006.

[55] J. Drake, P. Mehta, C. Miller, S. Moyer, R. Smith, and C. Valasek, "Browser Security Comparison: A Quantitative Approach," Accuvant Labs, Tech. Rep., 2011.

[56] S. Egelman, J. Tsai, L. F. Cranor, and A. Acquisti, "Timing is everything? the effects of timing and placement of online privacy indicators," in *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*, 2009.

[57] S. Egelman, L. F. Cranor, and J. Hong, "You've Been Warned: An empirical study of the effectiveness of web browser phishing warnings," in *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*, 2008.

[58] S. Egelman, A. P. Felt, and D. Wagner, "Choice Architecture and Smartphone Privacy: There's A Price For That," in *Workshop on the Economics of Information Security (WEIS)*, 2012.

[59] M. Eichin and J. Rochlis, "With microscope and tweezers: an analysis of the Internet virus of November 1988," May 1989.

[60] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of Android application security," in *Proceedings of the USENIX Security Symposium*, 2011.

[61] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*, 2009.

[62] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[63] "Enhanced JUnit." [Online]. Available: http://www.silvermark.com/Product/java/enhancedjunit/index.html

[64] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A Survey of Mobile Malware in the Wild," in *Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM)*, 2011.

[65] A. P. Felt, M. Finifter, J. Weinberger, and D. Wagner, "Diesel: Applying Privilege Separation to Database Access," in *Proceedings of the ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2011.

[66] A. P. Felt, "Issue 54006: Security: Extension history permission does not generate a warning," August 2010. [Online]. Available: http://code.google.com/p/chromium/issues/detail?id=54006

[67] T. Fraser, "LOMAC: Low Water-Mark Integrity Protection for COTS Environments," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2000.

[68] A. Fuchs, A. Chaudhuri, and J. Foster, "SCanDroid: Automated Security Certification of Android Applications," University of Maryland, Tech. Rep., 2009.

[69] G. J. Gaeth and J. Shanteau, "Reducing the Influence of Irrelevant Information on Experienced Decision Makers," *Organizational Behavior and Human Performance*, vol. 33, 1984.

[70] T. Garfinkel, B. Pfaff, and M. Rosenblum, "Ostia: A Delegating Architecture for Secure System Call Interposition," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2004.

[71] M. Gargenta, "Using Camera API," Marakana. [Online]. Available: http://marakana.com/forums/android/examples/39.html

[72] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Detecting Privacy Leaks in Android Applications," UC Davis, Tech. Rep., 2011.

[73] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications: Confining the wily hacker," in *Proceedings of the USENIX Security Symposium*, 1996.

[74] J. Golson, "Apple requires user permission before apps can access personal data in iOS 6," MacRumors. [Online]. Available: http://www.macrumors.com/2012/06/14/apple-requires-user-permission-before-apps-can-access-personal-data-in-ios-6/

[75] N. Good, R. Dhamija, J. Grossklags, S. Aronovitz, D. Thaw, D. Mulligan, and J. Konstan, "Stopping spyware at the gate: A user study of privacy, notice and spyware," in *Proceedings of the Symposium On Usable Privacy and Security (SOUPS)*, 2005.

[76] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy, "Verified security for browser extensions," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[77] D. Hackborn, "Re: List of private / hidden / system APIs?" [Online]. Available: http://groups.google.com/group/android-developers/msg/a9248b18cba59f5a

[78] S. E. Hallyn, "POSIX file capabilities: Parceling the power of root," 2007. [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-posixcap/index.html

[79] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[80] J. Howell and S. Schechter, "What you see is what they get: Protecting users from unwanted use of microphones, cameras, and other sensors," in *Proceedings of the Web 2.0 Security and Privacy Workshop (W2SP)*, 2010.

[81] S. Ibrahim, "Universal 1-Click Root App for Android Devices," August 2010. [Online]. Available: http://androidspin.com/2010/08/10/universal-1-click-root-app-for-android-devices/

[82] C. Jackson, "Block chrome-extension:// pages from importing script over non-https connections." [Online]. Available: http://code.google.com/p/chromium/issues/detail?id=29112

[83] A. Jakl, "Platform security," Symbian Resources, 2009. [Online]. Available: http://www.symbianresources.com/tutorials/general.php#security

[84] C. Jensen, C. Potts, and C. Jensen, "Privacy practices of Internet users: Self-reports versus observed behavior," in *Proceedings of the International Journal of Human-Computer Studies*, 2005.

[85] Ka-Ping Yee, "Secure Interaction Design and The Principle of Least Authority," in *Proceedings of the CHI Workshop on Human-Computer Interaction and Security Systems*, 2003.

[86] P. A. Karger, U. Roger, and R. Schell, "Multics security evaluation: Vulnerability analysis," HQ Electronic Systems Division: Hanscom AFB, MA. URL: http://csrc.nist.gov/publications/history/karg74.pdf, Tech. Rep., 1974.

[87] P. A. Karger and R. R. Schell, "Thirty years later: Lessons from the multics security evaluation," in *Annual Computer Security Applications Conference (ACSAC*, 2002.

[88] R. Karim, M. Dhawan, V. Ganapathy, and C. chiech Shan, "An Analysis of the Mozilla Jetpack Extension Framework," in *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP)*, 2012.

[89] P. G. Kelley, M. Benisch, L. F. Cranor, and N. Sadeh, "When are users comfortable sharing locations with advertisers?" in *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, 2011.

[90] P. G. Kelley, S. Consolovo, L. F. Cranor, J. Jung, N. Sadeh, and D. Wetherall, "A Conundrum of Permissions: Installing Applications on an Android Smartphone," in *Proceedings of the Workshop on Usable Security (USEC)*, 2012.

[91] J. King, A. Lampinen, and A. Smolen, "Privacy: Is There An App for That?" in *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2011.

[92] J. F. Koegel, R. M. Koegel, Z. Li, and D. T. Miruke, "A security analysis of VAX VMS," in *Proceedings of the ACM annual conference on The range of computing : mid-80's perspective: mid-80's perspective*, ser. ACM '85. ACM, 1985, pp. 381–386.

[93] A. Krishnamurthy, A. Mettler, and D. Wagner, "Fine-grained privilege separation for web applications," in *Proceedings of the International Conference on World Wide Web (WWW)*, 2010.

[94] M. Krohn, P. Efstathopoulos, C. Frey, F. Kaashoek, E. Kohler, D. Mazières, R. Morris, M. Osborne, S. VanDeBogart, and D. Ziegler, "Make Least Privilege a Right (Not a Privilege)," in *Proceedings of the Conference on Hot Topics in Operating Systems*, 2005.

[95] J. Krosnick and D. Alwin, "An evaluation of a cognitive theory of response-order effects in survey measurement," *Public Opinion Quarterly*, vol. 51, no. 2, Summer 1987.

[96] I. Krstic, "System Security on the One Laptop Per Child's XO Laptop: The Bitfrost security problem." [Online]. Available: http://dev.laptop.org/git/security/tree/bitfrost.txt

[97] P. Kumaraguru and L. F. Cranor, "Privacy Indexes: A Survey of Westin's Studies," Carnegie Mellon University CMU-ISRI-5-138, Tech. Rep., 2015.

[98] S. Lederer, J. Mankoff, and A. K. Dey, "Who wants to know what when? privacy preference determinants in ubiquitous computing," in *Proceedings of the ACM CHI extended abstracts on Human factors in computing systems*, 2003.

[99] W.-M. Lee, "SMS messaging in Android," MobiForge. [Online]. Available: http://mobiforge.com/developing/story/sms-messaging-android

[100] R. Leung, L. Findlater, J. McGrenere, P. Graf, and J. Yang, "Multi-Layered Interfaces to Improve Older Adults' Initial Learnability of Mobile Applications," *ACM Transactions on Accessible Computing (TACCESS)*, 2010.

[101] H. M. Levy, *Capability-Based Computer Systems*. Digital Press, 1984. [Online]. Available: http://www.cs.washington.edu/homes/levy/capabook/

[102] P. Lewis, "An introduction to shaders," HTML5 Rocks Tutorials. [Online]. Available: http://www.html5rocks.com/en/tutorials/webgl/shaders/

[103] L. Liu, X. Zhang, G. Yan, and S. Chen, "Chrome Extensions: Threat Analysis and Countermeasures," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[104] R. S. Liverani and N. Freeman, "Abusing Firefox Extensions," Defcon17.

[105] B. Livshits, J. Whaley, and M. S. Lam, "Reflection Analysis for Java," in *Proceedings of the Asian Symposium on Programming Languages and Systems*, 2005.

[106] I. Lunden, "Google Play About To Pass 15 Billion App Downloads?" Tech Crunch, May 2012. [Online]. Available: http://techcrunch.com/2012/05/07/google-play-about-to-pass-15-billion-downloads-pssht-it-did-that-weeks-ago/

[107] W. A. Magat, W. K. Viscusi, and J. Huber, "Consumer Processing of Hazard Warning Information," *Journal of Risk and Uncertainty*, vol. 1, 1988.

[108] L. Magid, "It pays to read license agreements," 2005. [Online]. Available: http://www.pcpitstop.com/spycheck/eula.asp

[109] G. McCluskey, "Using Java Reflection," Sun Developer Network, 1998. [Online]. Available: http://java.sun.com/developer/technicalArticles/ALT/Reflection/

[110] A. Mikhailovsky, K. V. Gavrilenko, and A. Vladimirov, "The Frame of Deception: Wireless Man-in-the-Middle Attacks and Rogue Access Points Deployment," InformIT, 2004. [Online]. Available: http://www.informit.com/articles/article.aspx?p=353735&seqNum=7

[111] S. Motiee, K. Hawkey, and K. Beznosov, "Do Windows Users Follow the Principle of Least Privilege? Investigating User Account Control Practices," in *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2010.

[112] K. Mueller and K. Butler, "Flex-P: Flexible Android Permissions," IEEE Symposium on Security and Privacy, Poster Session, 2011.

[113] D. Murray and S. Hand, "Privilege separation made easy: trusting small libraries not big processes," in *Proceedings of the European Workshop on System Security (EUROSEC)*, 2008.

[114] I. Muslukhov, Y. Boshmaf, C. Kuo, J. Lester, and K. Beznosov, "Understanding Users' Requirements for Data Protection in Smartphones," in *Proceedings of the ICDE Workshop on Secure Data Management on Smartphones and Mobiles*, 2012.

[115] E. Nasi, "Exploiting capabilities: Parcel root power, the dark side of capabilities," 2010.

[116] M. Nauman, S. Khan, M. Alam, and X. Zhang, "Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints," in *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.

[117] C. Pacheco and M. Ernst, "Eclat: Automatic generation and classification of test inputs," *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[118] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2007.

[119] C. Pacheco and M. Ernst, "Randoop." [Online]. Available: http://code.google.com/p/randoop/

[120] G. Paller, "Dedexer," `http://dedexer.sourceforge.net`.

[121] N. Provos, "Improving Host Security with System Call Policies," in *Proceedings of the USENIX Security Symposium*, 2003.

[122] N. Provos, M. Friedl, and P. Honeyman, "Preventing Privilege Escalation," in *Proceedings of the USENIX Security Symposium*, 2003.

[123] G. Richards, C.Hammer, B. Burg, and J. Vivek, "The Eval that Men Do: A Large-scale Study of the Use of Eval in JavaScript Applications," in *Proceedings of the European Conference on Object-Oriented Programming*, 2012.

[124] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Commun. ACM*, vol. 17, no. 7, pp. 365–375, 1974.

[125] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[126] A. Sacco, "How to Manage BlackBerry Application Permissions," 2011. [Online]. Available: http://www.cio.com/article/684233/How_to_Manage_BlackBerry_Application_Permissions

[127] J. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," in *Proceedings of the IEEE*, vol. 63, 1975.

[128] J. H. Saltzer, "Protection and the control of information sharing in Multics," *Commun. ACM*, vol. 17, no. 7, pp. 388–402, July 1974.

[129] R. Saltzman and A. Sharabani, "Active Man in the Middle Attacks: A Security Advisory," IBM, Tech. Rep., 2009.

[130] J. Sawin and A. Rountev, "Improving static resolution of dynamic class loading in Java using dynamically gathered environment information," *Automated Software Eng.*, vol. 16, pp. 357–381, June.

[131] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, "The Emperor's New Security Indicators," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[132] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney, "Model-carrying code: a practical approach for safe execution of untrusted applications," in *Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.

[133] S. Sen and D. Lerman, "Why are you telling me this? An examination into negative consumer reviews on the web," *Journal of Interactive Marketing*, vol. 21, 2007.

[134] N. Seriot, "iPhone Privacy," *Black Hat DC*, 2010.

[135] B. Shneiderman, "Promoting universal usability with multi-layer interface design," in *Proceedings of the Conference on Universal Usability (CUU)*, 2003.

[136] J. H. Simonetti, "How To Use Sky Image Processor (SIP 2.20)." [Online]. Available: http://www.phys.vt.edu/~jhs/SIP/howto.html

[137] R. N. Srinivas, Java World, 2000. [Online]. Available: http://www.javaworld.com/jw-07-2000/jw-0728-security.html

[138] B. Sterne and A. Barth, "Content security policy," W3C. [Online]. Available: https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html

[139] D. W. Stewart and I. M. Martin, "Intended and Unintended Consequences of Warning Messages: A Review and Synthesis of Empirical Research," *Journal of Public Policy Marketing*, vol. 13, no. 1, 1994.

[140] M. Stiegler and M. Miller, "A Capability Based Client: The DarpaBrowser," 2002. [Online]. Available: http://www.combex.com/papers/darpa-report/html/index.html

[141] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor, "Crying Wolf: An Empirical Study of SSL Warning Effectiveness," in *Proceedings of the USENIX Security Symposium*, 2009.

[142] H. Taylor, "Most People are "Privacy Pragmatists" Who, While Concerned about Privacy, Will Sometimes Trade It Off for Other Benefits," Harris Interactive, March 2003.

[143] The Android Open Source Project, "Dalvik executable format," 2007. [Online]. Available: http://source.android.com/tech/dalvik/dex-format.html

[144] S. Thurm and Y. I. Kane, "Your apps are watching you," 2010. [Online]. Available: http://online.wsj.com/article/SB10001424052748704694004576020083703574602.html

[145] J. Tsai, S. Egelman, L. Cranor, and A. Acquisti, "The effect of online privacy information on purchasing behavior: An experimental study," in *Workshop on the Economics of Information Security (WEIS)*, 2007.

[146] T. Vennon and D. Stroop, "Threat Analysis of the Android Market," SMobile Systems, Tech. Rep., 2010.

[147] T. Vidas, N. Christin, and L. Cranor, "Curbing Android Permission Creep," in *Proceedings of the Workshop on Web 2.0 Security and Privacy (W2SP)*, 2011.

[148] D. Wagner and D. Tribble, "A Security Analysis of the Combex DarpaBrowser Architecture," March 4 2002. [Online]. Available: http://www.combex.com/papers/darpa-review/security-review.html

[149] S. Wagner, J. Jurgens, C. Koller, and P. Trischberger, "Comparing Bug Finding Tools with Reviews and Tests," *Lecture Notes in Computer Science*, 2005.

[150] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993.

[151] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: practical capabilities for UNIX," in *Proceedings of the USENIX Security Symposium*, 2010.

[152] J. Weinberger, A. Barth, and D. Song, "Towards Client-side HTML Security Policies," in *Proceedings of the Workshop on Hot Topics on Security (HotSec)*, 2011.

[153] J. Wiese, P. G. Kelley, L. F. Cranor, L. Dabbish, J. I. Hong, and J. Zimmerman, "Are you close with me? are you nearby?: investigating social groups, closeness, and willingness to share," in *Proceedings of the International Conference on Ubiquitous Computing (UbiComp)*, 2011.

[154] S. Willison, "Understanding the Greasemonkey vulnerability." [Online]. Available: http://simonwillison.net/2005/Jul/20/vulnerability/

[155] M. S. Wogalter, "Communication-Human Information Processing (C-HIP) Model," in *Proceedings of the Handbook of Warnings*. Lawrence Erlbaum Associates, 2006, pp. 51–61.

[156] ——, "Purpose and scope of warnings," in *Proceedings of the Handbook of Warnings*. Lawrence Erlbaum Associates, 2006.

[157] C. Wuest and E. Florio, "Firefox and Malware: When Browsers Attack," Symantec, Tech. Rep., 2009.

[158] Z. E. Ye, S. Smith, and D. Anthony, "Trusted paths for browsers," *ACM Transactions on Information and System Security (TISSEC)*, 2005.

[159] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[160] F. Zhu and X. Zhang, "Impact of Online Consumer Reviews on Sales: The Moderating Role of Product and Consumer Characteristics," *Journal of Marketing*, vol. 74, 2010.