How to Ask For Permission

Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, David Wagner
{*apf, egelman, finifter, devdatta, daw*}*@cs.berkeley.edu*
University of California, Berkeley

Abstract

Application platforms provide applications with access to hardware (e.g., GPS and cameras) and personal data. Modern platforms use permission systems to protect access to these resources. The nature of these permission systems vary widely across platforms. Some platforms obtain user consent as part of installation, while others display runtime consent dialogs. We propose a set of guidelines to aid platform designers in determining the most appropriate permission-granting mechanism for a given permission. We apply our proposal to a smartphone platform. A preliminary evaluation indicates that our model will reduce the number of warnings presented to users, thereby reducing habituation effects.

1 Introduction

In traditional platforms, applications inherit all of the privileges of the users who execute them. In modern platforms, applications run with few privileges by default and can only obtain additional privileges via the platform's permission system. For example, an Android application can gain access to the user's geolocation with the `ACCESS_FINE_LOCATION` permission. We see permission systems in smartphones (e.g., Android, iOS, Windows Phone), browsers (e.g., Chrome extensions and web apps), web platforms (e.g., the Facebook Platform and the Twitter API), and desktop operating systems (e.g., Mac App Store and Windows 8 Metro).

Typically, platforms with permission systems involve the user in the permission-granting process in order to (1) inform the user about potential risks and (2) engage the user in security and privacy decisions. The quality of the user's experience with permissions is fundamental to successfully informing and empowering users. A platform designer has to consider diverse aspects of the user experience when creating the permission-granting process. When should the platform ask the user for consent? What types of permissions need consent? How much user interaction is too much? We aim to answer these questions by discussing the strengths and weaknesses of the different permission-granting mechanisms that are available to platform designers.

Currently, there is no consensus on the best way to design a permission system. This is evidenced by the wide disparity in the user experience for two popular smartphone platforms: Android and iOS. Android presents 124 permissions to users with install-time warnings, whereas iOS obtains user consent for two permissions with runtime confirmation dialogs. A number of research proposals argue for the use of trusted UI in permission systems [8, 9, 12, 15], 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.

Contributions. We contribute the following:

- We enumerate the permission-granting mechanisms 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.
- 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.

Our work is primarily targeted at the designer of a new permission system for a platform that does not have an application review process. A platform with an application review process might also make use of our recommendations when it is not possible or appropriate to review a permission; that decision is beyond our scope.

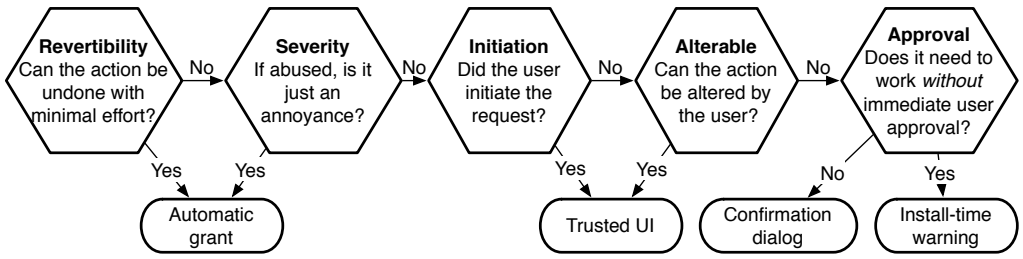


Figure 1: A guide to selecting between the different permission-granting mechanisms.

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 [1, 10] 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.

2.1 Conserve User Attention

Human attention is a shared, finite resource [1]. 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 for Windows UAC dialogs, Android install-time warnings, and browser security warnings [3, 6, 11, 13]. Users pay less attention to each subsequent warning, and this effect carries over to other, similar warnings [2].

Principle 1: Conserve user attention, utilizing it only for permissions that have severe consequences.

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 her primary goal of installing the application.

Users have low motivation to consider interruptive security mechanisms, which encourages satisficing. Concretely, this means that the user will click through a dialog box that interrupts her main task without fully understanding the consequences. Good et al. found that users dismiss or ignore interruptive warnings [7]. In a study of Windows UAC consent dialogs, Motiee et al. observed that most study participants clicked through illegitimate UAC prompts while completing tasks [11].

Principle 2: When possible, avoid interrupting the user's primary task with explicit security decisions.

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 1 summarizes our proposed selection criteria.

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.

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 auditing mechanisms allow users to identify and uninstall abusive applications. Figure 2 provides examples of auditing mechanisms for automatically-granted permissions.

(a) The user can see which application last changed the wallpaper and revert.

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

Figure 2: Examples of auditing for automatic grants.

(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 3: A photo chooser.

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.

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

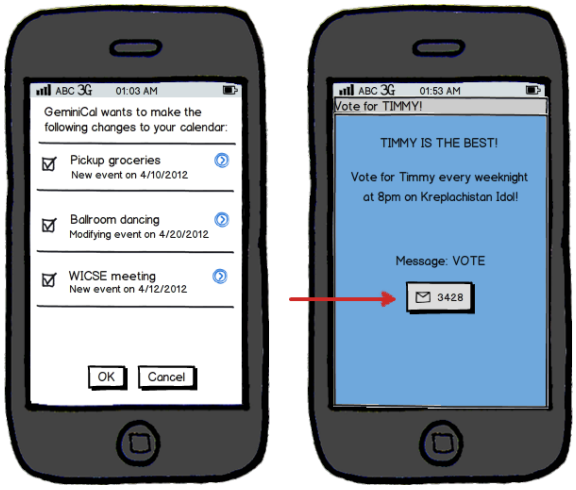
3.2 Trusted UI

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 [14], access control gadgets [12], and sensor-access widgets [8]. 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 3 shows an example photo chooser as trusted UI. Web browsers and Windows 8 Metro currently rely on choosers for providing file access.
- *Review screens* allow users to review, accept, or modify proposed changes. For example, before adding events to a calendar, a review screen allows the user to see and optionally edit the events (Figure 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 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 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 [12]. Motivation and positive expectations lead to optimizing.

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



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

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

Figure 4: Item review screen and permission-granting button.

(a) iOS's location or notifications dialog.

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

Figure 5: Runtime consent dialogs.

this use case because the application performs the action in response to external state, rather than the user.[1]

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).

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 5(a)), or it can include customized dialogs for different actions (e.g., Figure 5(b)).

[1]Roesner et al. proposed embedded buttons with text such as "permanent access" [12]. We do not view these as trusted UI because they are not part of the normal workflow of an application.

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.

Weaknesses. Runtime consent dialogs encourage satisficing because they are interruptive and repetitive.

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.

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 [6]. Compounding their monotony, install-time warnings also suffer from being similar to EULAs. Users typically ignore EULAs [7], and their habituation to EULAs extends to other types of indicators that resemble EULAs [2]. This implies that install-time warnings may not adequately capture users' attention.

Users may incorrectly believe that an application cannot use a permission granted during installation until the user confirms this a second time during runtime [12, 6].

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

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 [5, Section 4.1.5]. 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.

4 Applying Our Guidelines

We systematically assigned permission-granting mechanisms to smartphone 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.

4.1 Permission Assignment

To perform the assignment, we first created a set of platform-neutral smartphone permissions by combining Android permissions, Windows Phone 7 capabilities, iOS prompts, and the Mozilla WebAPI. After grouping redundant permissions and dividing broad permissions,[2] we arrived at 83 platform-neutral permissions. We then considered each permission individually. We devised auditing and trusted UI mechanisms as appropriate; Figures 2, 3, and 4 illustrate some of our proposed mechanisms. As part of our analysis, we surveyed 3,115 smartphone users to identify low-severity permissions [4].

Based on our analysis, 55% of permissions can be automatically granted, 23% can be represented with trusted UI, 16% require runtime consent dialogs, and 6% would need install-time warnings. We find that runtime consent dialogs and install-time warnings cannot be completely avoided, given that we do not wish to disable application features. However, our assignment demonstrates that the majority of permissions can be handled with non-interruptive permission-granting mechanisms.

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 still camera permission. After examining applications that take photos, we found that past proposals for controlling camera access with trusted UI [12, 8] would not satisfy many popular applications:

1. 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. 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.

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.

[2]For example, we grouped Android's "automatically start at boot" and "make application always run" permissions into "run all the time."

4.2 Preliminary Evaluation

We reviewed the twenty most popular free iPhone applications to evaluate how our proposed permission system would impact the user experience. We manually tested the applications to match their behavior to the 83 API calls discussed in Section 4.

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.

We also 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.

These results suggest that our approach could plausibly provide a workable basis for asking for permission, without triggering habituation.

5 Conclusion

Current third-party application platforms do not consider risk or usability when determining which mechanism to use when applications request permissions. In fact, most platforms simply use the same mechanism for all permissions. We propose a new model (Figure 1) to help platform developers choose the most appropriate mechanism for a given permission. While our model seems like a plausible direction, further research is still needed to perform a comprehensive usability evaluation of a platform that follows our model.

Acknowledgements

This material is based upon work supported by a Facebook Fellowship, National Science Foundation Graduate Research Fellowships, NSF grant CNS-1018924, a gift from Google, and the Intel Science and Technology Center for Secure Computing. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the entities that provided funding.

References

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

[2] 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.

[3] 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 26th SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 2008.

[4] A. P. Felt, S. Egelman, and D. Wagner. I've Got 99 Problems, But Vibration Ain't One: A Survey of Smartphone Users' Concerns. Technical Report UCB/EECS-2012-70, May 2012.

[5] 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 Smartphones and Mobile Devices (SPSM)*, 2011.

[6] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[7] 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.

[8] 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 & Privacy Workshop (W2SP)*, 2010.

[9] 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.

[10] J. Krosnick and D. Alwin. An evaluation of a cognitive theory of response-order effects in survey measurement. *Public Opinion Quarterly*, 51(2):201–219, Summer 1987.

[11] 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.

[12] 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 IEEE Symposium on Security and Privacy*, 2012.

[13] 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.

[14] D. Wagner and D. Tribble. A Security Analysis of the Combex DarpaBrowser Architecture, March 4 2002. http://www.combex.com/papers/darpa-review/security-review.html.

[15] Z. E. Ye, S. Smith, and D. Anthony. Trusted paths for browsers. *ACM Transactions on Information and System Security (TISSEC)*, 2005.