CHAPTER THIRTEEN

Guidelines and Strategies for Secure Interaction Design

KA-PING YEE

ALTHOUGH A RELIABLE, USABLE AUTHENTICATION METHOD IS ESSENTIAL, it is far from the only human interface concern. After a user signs in to a system, the system has to carry out the user's wishes correctly in order to be considered secure. The question of secure interaction design, addressed in this and the other chapters in this part of the book, is:

How can we design a computer system to protect the interests of its legitimate user?

To give you a sense of how important it is to look beyond authentication, consider some of today's most serious security problems. Viruses are a leading contender, with email viruses making up a large part. Spyware is growing into a nightmare for home users and IT staff. Identity theft is becoming widespread, perpetrated in part through "phishing" scams in which forged email messages entice people to give away private information. None of these problems is caused by defeating a login mechanism. They would be better described as failures of computers to behave as their users expect.

This chapter suggests some guidelines for designing and evaluating usable secure software and proposes two strategies for getting security and usability to work in harmony: *security by designation* and *user-assigned identifiers*. I'll begin by providing a little background for our discussion, then present the guidelines and strategies, and finally look at real design problems to show how these strategies can be applied in practice.

Introduction

This section introduces the topic of secure interaction design, touching briefly on mental models, the apparent conflict between security and usability, and the overall issues underlying interaction design.

Mental Models

For software to protect its user's interests, its behavior should be consistent with the user's expectations. Therefore, designing for security requires an understanding of the *mental model*—the user's idea of how the system works. Attempting to make security decisions completely independent of the user is ultimately futile: if security really were independent, the user would lack the means to predict and understand the consequences of her actions. On the other hand, users should not be expected to speak the language of security experts or think in terms of the security mechanisms to get their work done safely. Thus, usable secure systems should enforce security decisions based on the user's actions while allowing those actions to be expressed in familiar ways.

A common error in security thinking is to classify an entity as "trusted" or "untrusted" without carefully defining *who* is doing the trusting and exactly *what* they trust the entity to do. To ignore these questions is to ignore the mental model. For example, suppose that a user downloads a file-deletion program. The program has been carefully tested to ensure that it reliably erases files, leaving no trace. Is the program secure or not? If the program is advertised as a game and it deletes the user's files, that would be a security violation. However, if the user intends to erase files containing sensitive information, the program's *failure* to delete them would be a security violation. The only difference is in the expectations. A digital signature asserting who created the program[1] provides no help. The correctness of a program provides no assurance that it is secure, and the presence of a digital signature provides no assurance that it is either correct *or* secure.

Sources of Conflict

At the heart of the apparent conflict between security and usability is the idea that security makes operations harder, yet usability makes operations easier. Although this is usually true, it's imprecise. Security isn't about making *all* operations difficult; it's about restricting access to operations with undesirable effects. Usability isn't about making *all* operations easy, either; it's about improving access to operations with desirable effects. Tension between the two arises to the extent that a system is unable to determine whether a particular result is desirable. Security and usability come into harmony when a system correctly interprets the user's desires.

1 Today's code-signing schemes are designed to provide reliable verification that a program came from a particular source. They don't warrant what programs are supposed to do. They usually don't even specify the author of a program, merely the entity whose key was used to sign it.

Presenting security to users as a secondary task also promotes conflict. Some designs assume that users can be assigned extra homework: users are expected to install patches, run virus scanners, set up firewalls, and/or check certificates in order to keep their computers secure. But most people don't buy computers and take them home so that they can curl up for a nice evening of firewall configuration. They have better things to do with their computers, like keeping in touch with their friends, writing documents, playing music, or organizing their lives. Users are unlikely to perform extra security tasks and may even circumvent security measures to make their main task more comfortable.

As Diana Smetters and Rebecca Grinter have suggested,[2] security goals should be closely integrated with the workflow of the main task to yield *implicit security*. Extracting information about security expectations from the user's normal interactions with the interface enables us to minimize or eliminate the need for secondary security tasks.

Iterative Design

Security and usability are qualities that apply to a whole system, not features that can be tacked on to a finished product. It's regrettably common to hear people speak of "adding security features," even though they would find the idea of adding a "correctness feature" absurd. Trying to add security after the fact is rarely effective and often harms usability as well. Likewise, although usability is also often described in terms of particular features (such as icons, animations, themes, or widgets), *interaction design* is much deeper than visual appearance. The effectiveness of a design is affected by whether work flows smoothly, whether the symbols and concepts make sense to the user, and whether the design fits the user's mental model of actions and their consequences.

Instead of adding security or usability as an afterthought, it's better to design software in iterations consisting of three basic phases: analysis of needs, then design, then testing. After a round of testing, it's time to analyze the results to find out what needs to be improved, then apply these discoveries to the next round of design, and so on. With each cycle, prototypes become more elaborate and polished as they approach product quality. This advice is nothing new; software engineers and usability engineers have advocated iterative design for many years. However, iterative design is particularly essential for secure software, because security and usability design choices affect each other in ways that are difficult to predict and are best understood through real tests.

Every piece of software ultimately has a human user, even if that user is sometimes a system administrator or a programmer. Therefore, attention to usability concerns is always necessary to achieve true security. The next time someone tells you that his product is secure, you might want to ask, "How much user testing have you done?"

2 Diana Smetters and Rebecca Grinter, "Moving from the Design of Usable Secure Technologies to the Design of Useful Secure Applications," *Proceedings of the 2002 Workshop on New Security Paradigms* (New York: ACM Press, 2002).

Permission and Authority

Much of the following discussion concerns the management of authorities. By an *authority*, I mean the power to make something happen, in contrast to *permission*, which refers to access as represented by settings in a security mechanism.[3]

For example, suppose that Alice's computer records and enforces the rule that only Alice can read the files in her home directory. Alice then installs a program that serves up her files on the Web. If she gives Bob the URL to her web server, she is granting Bob the authority to read her files even though Bob has no such permission in Alice's system. From the system's perspective, restricted permissions are still being enforced, because Alice's files are accessible only to Alice's programs. It's important to keep in mind the difference between permission and authority because so many real-world security issues involve the transfer of authority independent of permissions.

Design Guidelines

Having established this background, we are now ready to look at the main challenge of secure interaction design: minimizing the likelihood of undesired events while accomplishing the user's intended tasks correctly and as easily as possible. Let's dissect that general aim into more specific guidelines for software behavior.

Minimizing the risk of undesired events is a matter of controlling *authorization*. Limiting the authority of other parties to access valuable resources protects those resources from harm. The authorization aspect of the problem can be broken down into five guidelines:

1. Match the most comfortable way to do tasks with the least granting of authority.
2. Grant authority to others in accordance with user actions indicating consent.
3. Offer the user ways to reduce others' authority to access the user's resources.
4. Maintain accurate awareness of others' authority as relevant to user decisions.
5. Maintain accurate awareness of the user's own authority to access resources.

Accomplishing the user's intended tasks correctly depends on good *communication* between the user and the system. The user should be able to convey his desires to the system accurately and naturally. I'll discuss the following additional guidelines concerning the communication aspect of the problem:

6. Protect the user's channels to agents that manipulate authority on the user's behalf.
7. Enable the user to express safe security policies in terms that fit the user's task.
8. Draw distinctions among objects and actions along boundaries relevant to the task.

3 These definitions of permission and authority are due to Mark S. Miller and Jonathan Shapiro. See Mark S. Miller and Jonathan S. Shapiro, "Paradigm Regained: Abstraction Mechanisms for Access Control," in Vijay A. Saraswat (ed.), *Proceedings of the 8th Asian Computing Science Conference, Lecture Notes in Computer Science* 2896, (Heidelberg: Springer-Verlag, 2003); *http://erights.org/talks/asian03/*.

9. Present objects and actions using distinguishable, truthful appearances.
10. Indicate clearly the consequences of decisions that the user is expected to make.

These guidelines are built on straightforward logic and are gleaned from the experiences of security software designers. They are not experimentally proven, although the reasoning and examples given here should convince you that violating these guidelines is likely to lead to trouble. I'll present each guideline along with some questions to consider when trying to evaluate and improve designs. Strategies to help designs follow some of these guidelines are provided in the second half of this chapter.

Authorization

1. Match the most comfortable way to do tasks with the least granting of authority.

What are the typical user tasks?
What is the user's path of least resistance for each one?
What authorities are given to software components and other users when the user follows this path?
How can the safest ways of accomplishing a task be made more comfortable, or the most comfortable ways made safer?

In 1975, Jerome Saltzer and Michael Schroeder wrote a landmark paper on computer security[4] proposing eight design principles; their *principle of least privilege* demands that we grant processes the minimum privilege necessary to perform their tasks. This guideline combines that principle with an acknowledgment of the reality of human preferences: when people are trying to get work done, they tend to choose methods that require less effort, are more familiar, or appear more obvious. Instead of fighting this impulse, use it to promote security. Associate greater risk with greater effort, less conventional operations, or less visible operations so that the user's natural tendency leads to safe operation.

A natural consequence of this guideline is to default to a lack of access and to require actions to grant additional access, instead of granting access and requiring actions to shut it off. (Saltzer and Schroeder called this *fail-safe defaults*.) Although computer users are often advised to change network parameters or turn off unnecessary services, the easiest and most obvious course of action is not to reconfigure anything. Wendy Mackay has identified many barriers to user customization: customization takes time, it can be hard to figure out, and users don't want to risk breaking their software. [5]

4 Jerome Salzter and Michael Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the 4th Symposium on Operating System Principles* (ACM Press, 1973); *http://web.mit.edu/Saltzer/www/publications/protection/*.

5 Wendy Mackay, "Users and Customizable Software: A Co-Adaptive Phenomenon," (Ph.D. Thesis, Massachusetts Institute of Technology, 1990); *http://www-ihm.lri.fr/~mackay/pdffiles/MIT.thesis.A4.pdf*.

Consider Microsoft Internet Explorer's handling of remote software installation in the context of this guideline. Just before Internet Explorer runs downloaded software, it looks for a publisher's digital signature on the software and displays a confirmation window like the one shown in Figure 13-1. In previous versions of this prompt, the default choice was "Yes". As this guideline would suggest, the default is now "No", the safer choice. However, the prompt offers an option to "Always trust content" from the current source, but no option to *never* trust content from this source. Users who choose the safer path are assured a continuing series of bothersome prompts.

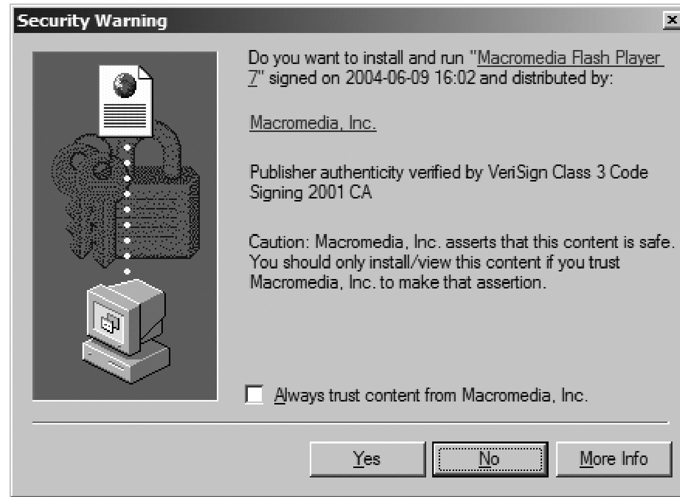


FIGURE 13-1. Internet Explorer 6.0 displays a software certificate

Regardless of the default settings, the choice being offered is poor: the user must either give the downloaded program complete access to all the user's resources, or not use the program at all. The prompt asks the user to be sure he trusts the distributor before proceeding. But if the user's task requires using the program, the most comfortable path is to click "Yes" without thinking. It will always be easier to just choose "Yes" than to choose "Yes" after researching the program and its origins. Designs that rely on users to assess the soundness of software are unrealistic. The principle of least privilege suggests that a mechanism for running programs with less authority would be better.

2. Grant authority to others in accordance with user actions indicating consent.

When does the system authorize software components or other users to access the user's resources?
What user actions trigger these transfers of authority?
Does the user consider these actions to indicate consent to such access?

The user's mental model of the system includes a set of expectations about *who* can do *what*. To prevent unpleasant surprises, we should ensure that other parties don't gain access that exceeds the user's expectations. When a program or another user is granted access to the user's resources, that granting should be related to some user action. If another party gains access without user action, the user lacks any opportunity to update his mental model to include knowledge of the other party's access.

The authorizing action doesn't have to be a security setting or a response to a prompt about security; ideally, it shouldn't feel like a security task at all. It should just be some action that the user associates with the granting of that power. For example, if we ask users whether they expect that double-clicking on a Microsoft Word document would give Word access to its contents, and the vast majority say yes, the double-click alone is sufficient to authorize Word to access the document.

In other situations, a user action can be present but not understood to grant the power it grants. This is the case when a user double-clicks on an email attachment and gets attacked by a nasty virus. The double-click is expected to open the attachment for viewing, not to launch an unknown program with wide-ranging access to the computer.

3. Offer the user ways to reduce others' authority to access the user's resources.

What types of access does the user grant to software components and other users?
Which of these types of access can be revoked?
How can the interface help the user to find and revoke such access?

After granting authorities, the user needs to be able to take them back in order to retain control of the computer. Without the ability to revoke access, the user cannot simplify system behavior and cannot recover from mistakes in granting access.

Closing windows is an example of a simple act of revocation that users understand well. When users close a document window, they expect the application to make no further changes to the document. This reduces the number of variables they have to worry about and lets them get on with other tasks.

Lack of revocability is a big problem when users are trying to uninstall software. The process of installing an application or a device driver usually provides no indication of what resources are given to the new software, what global settings are modified, or how to restore the system to a stable state if the installation fails. Configuration changes can leave the system in a state where other software or hardware no longer works properly. Microsoft Windows doesn't manage software installation or removal; it leaves these tasks up to applications, which often provide incomplete removal tools or no removal tools at all. Spyware exploits this problem: most spyware is designed to be difficult to track down and remove. An operating system with good support for revocability would maintain accurate records of installed software, allow the user to select unwanted programs, and cleanly remove them.

4. Maintain accurate awareness of others' authority as relevant to user decisions.

What kinds of authority can software components and other users hold?
Which kinds of authority impact user decisions with security consequences?
How can the interface provide timely access to information about such authorities?

To use any software safely, the user must be able to evaluate whether particular actions are safe, which requires accurate knowledge of the possible consequences. The consequences are bounded by the access that has been granted to other parties. Because

human memory is limited and fallible, expecting users to remember the complete history of authorizations is unrealistic. The user needs a way to refresh his mental model by reviewing which programs or other users have access to do which things.

Special attention must be paid to powers that continuously require universal trust, such as intercepting user input or manipulating the internal workings of running programs. For example, Microsoft Windows provides facilities enabling programs to record keystrokes and simulate mouse clicks in arbitrary windows. To grant such powers to another entity is to trust that entity completely with all of one's access to a system, so activation of these authorities should be accompanied by continuous notification.

Spyware exploits both a lack of consent to authorization and a lack of awareness of authority. Spyware is often included as part of other software that the user voluntarily downloads and wants to use. Even while the other software isn't being used, the spyware remains running in the background, invisibly compromising the user's privacy.

5. Maintain accurate awareness of the user's own authority to access resources.

What kinds of authority can tPGPkeyshe user hold?
How is the user informed of currently held authority?
How does the user come to know about acquisition of new authority?
What decisions might the user make based on his expectations of authority?

Users are also part of their own mental models. Their decisions can depend on their understanding of their own access. When users overestimate their authority, they may become vulnerable to unexpected risks or make commitments they cannot fulfill.

PayPal provides a good example of this problem in practice. When money is sent, PayPal sends the recipient an email message announcing "You've got cash!" Checking the account at the PayPal site will show the transaction marked "completed," as in Figure 13-2. If the sender of the money is buying an item from the recipient, the recipient would probably feel safe at this point delivering the item.

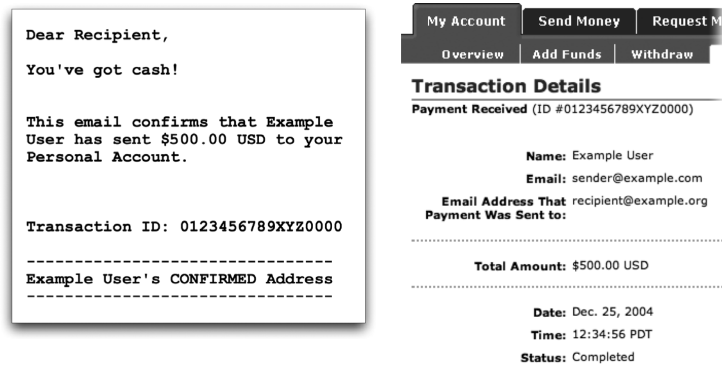


FIGURE 13-2. PayPal sends email notifying a recipient of "cash" and displays the details of a "completed" transaction

Unfortunately, PayPal's announcement generates false expectations. Telling the recipient that they've "got cash" suggests that the funds are concrete and under the recipient's control. Prior experience with banks may lead the recipient to expect that transactions clear after a certain period of time, just as checks deposited at most banks clear after a day or two. But although PayPal accounts look and act like bank accounts in many ways, PayPal's policy on payments[6] does not commit to any time limit by which payments become permanent; PayPal can still reverse the payment at any time. By giving the recipient a false impression of access, PayPal exposes the recipient to unnecessary risk.

Communication

6. Protect the user's channels to agents that manipulate authority on the user's behalf.

What agents manipulate authority on the user's behalf?
How can the user be sure that he is communicating with the intended agent?
How might the agent be impersonated?
How might the user's communication with the agent be intercepted or corrupted?

When someone uses a computer to interact in a particular world, there is a piece of software that serves as the user's agent in the world. For example, a web browser is the agent for interacting with the World Wide Web; a desktop GUI or command-line shell is the agent for interacting with the computer's operating system. If communication with that agent can be spoofed or corrupted, the user is vulnerable to an attack. In standard terminology, the user needs a *trusted path* for communicating with the agent.

The classic way to exploit this issue is to present a fake password prompt. If a web browser doesn't enforce a distinction between its own password prompts and prompt windows that web pages can generate, a malicious web site could use an imitation prompt to capture the user's password.

Techniques for preventing impersonation include designating reserved hotkeys, reserving areas of the display, and demonstrating privileged abilities. Microsoft Windows reserves the Ctrl-Alt-Delete key combination for triggering operating system security functions: no application program can intercept this key combination, so when users press it to log in, they can be sure that the password dialog comes from the operating system. Many web browsers reserve an area of their status bar for displaying an icon to indicate a secure connection. Trusted Solaris reserves a "trusted stripe" at the bottom of the screen for indicating when the user is interacting with the operating system.

Eileen Ye and Sean Smith have proposed adding flashing colored borders[7] to distinguish window areas controlled by the browser from those controlled by the remote site. The

6 PayPal, "Payments (Sending, Receiving, and Withdrawing) Policy" (Nov. 21, 2004); *http://www.paypal.com/cgi-bin/webscr?cmd=p/gen/ua/policy_payments-outside*.

7 Zishuang (Eileen) Ye and Sean Smith, "Trusted Paths for Browsers," *Proceedings of the 11th USENIX Security Symposium* (USENIX, 2002); *http://www.usenix.org/events/sec02/ye.html*.

borders constantly flash in a synchronized but unpredictable pattern, which makes them hard to imitate, but would probably annoy users. For password prompts, the Safari web browser offers a more elegant solution: the prompt drops out of the titlebar of the browser window and remains attached (see Figure 13-3) in ways that would be difficult for a web page script to imitate. Attaching the prompt to the window also prevents password prompts for different windows from being confused with each other.

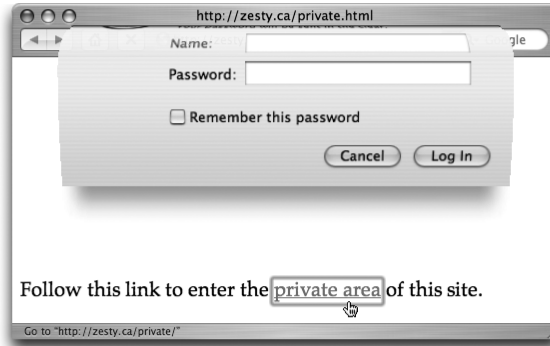


FIGURE 13-3. Password prompts in Safari fall out of the titlebar like a flexible sheet of paper and remain attached to the associated window

7. Enable the user to express safe security policies in terms that fit the user's task.

What are some examples of security policies that users might want enforced for typical tasks?
How can the user express these policies?
How can the expression of policy be brought closer to the task, ideally disappearing into the task itself?

If security policies are expressed using unfamiliar language or concepts unrelated to the task at hand, users will find it difficult to set a policy that corresponds to their intentions. When the security model doesn't fit, users may expose themselves to unnecessary risk just to get their tasks done.

For instance, one fairly common task is to share a file with a group of collaborators. Unix file permissions don't fit this task well. In a standard Unix filesystem, each file is assigned to one owner and one group. The owner can choose any currently defined group, but cannot define new groups. Granting access to a set of other users is possible only if a group is already defined to contain those users. This limitation encourages users to make their files globally accessible, because that's easier than asking the system administrator to define a new group.

8. Draw distinctions among objects and actions along boundaries relevant to the task.

At what level of detail does the interface allow objects and actions to be separately manipulated?
During a typical task, what distinctions between affected objects and unaffected objects does the user care about?
What distinctions between desired actions and undesired actions does the user care about?

Computer software systems consist of a very large number of interacting parts. To help people handle this complexity, user interfaces aggregate objects into manageable chunks: for example, bytes are organized into files, and files into folders. User interfaces also aggregate actions: downloading a web page requires many steps in the implementation, but for the user it's a single click. Interface design requires decisions about which distinctions to expose and hide. Exposing pointless distinctions generates work and confusion for the user; hiding meaningful distinctions forces users to take unnecessary risks.

On a Mac, for example, an application is shown as a single icon even though, at the system level, that icon represents a set of folders containing all the application's files. The user can install or remove the application by manipulating just that one icon. The user doesn't have to deal with the individual files, or risk separating the files by mistake. This design decision simplifies the user's experience by hiding distinctions that don't matter at the user level.

On the other hand, the security controls for web page scripts in Mozilla neglect to make important distinctions. Mozilla provides a way for signed scripts to gain special privileges,[8] but the only option for file access is a setting that grants access to all files. When the user is asked whether to grant that permission, there is no way to control which files the script is allowed to access. The lack of a boundary here forces the user to gamble the entire disk just to access one file.

9. Present objects and actions using distinguishable, truthful appearances.

How does the user identify and distinguish different objects and different actions?
In what ways can the means of identification be controlled by other parties?
What aspects of an object's appearance are under system control?
How can those aspects be chosen to best prevent deception?

In order to use a computer safely, the user needs to be able to identify the intended objects and actions when issuing commands to the computer. If two objects look indistinguishably similar, the user risks choosing the wrong one. If an object comes to have a misleading name or appearance, the user risks placing trust in the wrong object. The same is true for actions that have some representation in the user interface.

Identification problems can be caused by names or appearances that are hard to distinguish even if they aren't exactly the same. For example, in some typefaces, the lowercase "L" looks the same as the digit "1" or the uppercase "I", making some names virtually indistinguishable; later in this chapter, we'll look at a phishing attack that exploits just this ambiguity. Unicode adds another layer of complexity to the problem, because different character sequences can be displayed identically: an unaccented character followed by a combining accent appears exactly the same as a single accented character.

8 Jesse Ruderman, "Signed Scripts in Mozilla"; *http://www.mozilla.org/projects/security/components/signed-scripts.html*.

No interface can prevent other parties from lying. However, unlike objects in the real world, objects in computer user interfaces do not have primary control over their own appearances. The software designer has the freedom to choose any appearance for objects in the user interface. Objects can "lie" in their appearances only to the extent that the user interface relies upon them to present themselves. The designer must choose carefully what parts of the name or appearance are controlled by the system or controlled by the object, and uphold expectations about consistent parts of the appearance.

For instance, Microsoft Windows promotes the convention that each file's type is indicated by its extension (the part of the filename after the last period) and that an icon associated with the file type visually represents each file. The file type determines how the file will be opened when the icon is double-clicked. Unfortunately, the Windows Explorer defaults to a mode in which file extensions are hidden; in addition, executable programs (the most dangerous file type of all) are allowed to choose any icon to represent themselves. Thus, if a program has the filename *document.txt.exe* and uses the icon for a text file, the user sees a text file icon labeled *document.txt*. Many viruses have exploited this design flaw to disguise themselves as harmless files. By setting up expectations surrounding file types and also providing mechanisms to violate these expectations, Windows grants viruses the power to lie.

10. Indicate clearly the consequences of decisions that the user is expected to make.

What user decisions have security implications?
When such decisions are being made, how are the choices and their consequences presented?
Does the user understand the consequences of each choice?

When the user manipulates authorities, we should make sure that the results reflect what the user intended. Even if the software can correctly enforce a security policy, the policy being enforced might not be what was intended if the interface presents misleading, ambiguous, or incomplete information. The information needed to make a good decision should be available before the action is taken.

Figure 13-4 shows an example of a poorly presented decision. Prompts like this one are displayed by the Netscape browser when a web page script requests special privileges. (Scripts on web pages normally run with restricted access for safety reasons, although Netscape has a feature allowing them to obtain additional access with user consent.) The prompt asks the user to grant a privilege, but it doesn't describe the privilege to be granted, the length of time it will remain in effect, or how it can be revoked. The term "UniversalXPConnect" is almost certainly unrelated to the user's task. The checkbox labeled "Remember this decision" is also vague, because it doesn't indicate how the decision would be generalized—does it apply in the future to all scripts, all scripts from the same source, or repeated uses of the same script?

An interface can also be misleading or ambiguous in nonverbal ways. Many graphical interfaces use common widgets and metaphors, conditioning users to expect certain unspoken conventions. For example, a list of round radio buttons indicates that only one

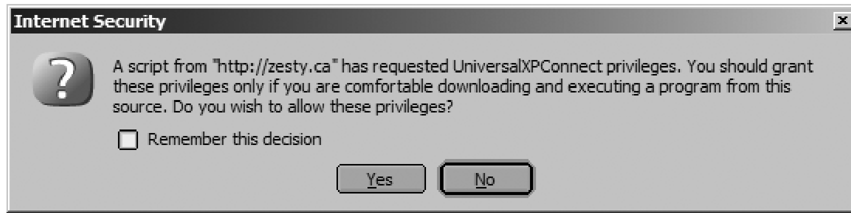


FIGURE 13-4. A script requests enhanced privileges in Netscape 7.2

of the options can be selected, whereas a list of square checkboxes suggests that any number of options can be selected. Visual interfaces also rely heavily on association between elements, such as the placement of a label next to a checkbox or the grouping of items in a list. Breaking these conventions causes confusion.

Design Strategies

Two strategies—security by designation and user-assigned identifiers—can be used to implement some of the aforementioned guidelines in a broad range of situations. After describing them, I'll propose ways to solve some everyday security problems using these strategies.

Security by Admonition and Security by Designation

Users often want to make use of things they do not completely trust. For example, it's reasonable for people to want to run programs or visit web pages without having to understand and audit their source code. Instead of trusting the unknown entity, users trust an agent (such as a secure operating system or a secure web browser) to protect their interests by placing constraints on the unknown entity. The agent's challenge is to determine the correct constraints.

Consider two contrasting styles of establishing these security constraints that I'll call *security by admonition* and *security by designation*. Security by admonition consists of providing notifications to which the user attends in order to maintain security. With security by designation, the user simultaneously designates an action and conveys the authority to perform the action.

The case of an operating system constraining a potentially dangerous program provides an example to illustrate these concepts. I'll use Venn diagrams to depict the space of the program's possible actions. In each run of the program, its actions form a path through this space, represented by a sequence of black dots. Arrows represent user actions that trigger these program actions. The rectangle delimits the actions that the system allows; it is a simple shape to signify the rigidity of a typical policy specification. The shaded region is the set of actions acceptable to the user; its shape is irregular to signify that the user's desires are probably complex and imprecisely specified. Figure 13-5 displays these notational conventions.

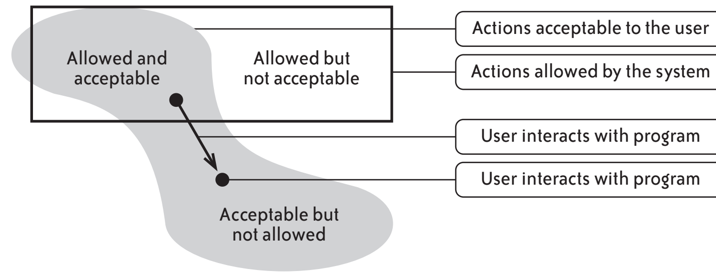


FIGURE 13-5. This Venn diagram of the space of program actions shows the notational conventions to be used in the next two figures

Security by admonition

Figure 13-6 shows a system that enforces a static, preestablished security policy. Because the policy is only finitely detailed, it can only roughly approximate the user's desires. Because the policy is static, it must accommodate a wide range of situations other than this particular run of the program. Thus, the solid box in Figure 13-6 includes large areas outside the shaded region. On Mac OS, Unix, and Microsoft Windows systems, programs usually run with the user's full authority, creating a vast discrepancy between the sets of allowed and acceptable actions. For example, even though I may start a program intending to edit only one file, the system allows it to modify or delete any of my files, send email in my name, upload my files to the Internet, and so on.

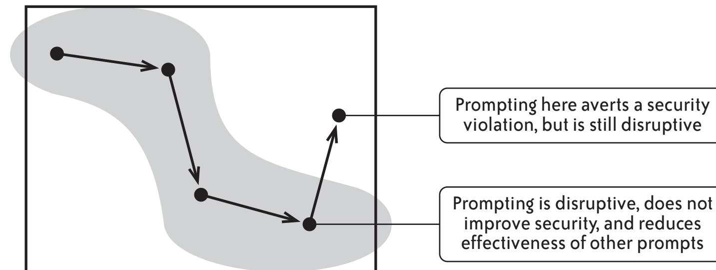


FIGURE 13-6. With security by admonition, the system has to guess when to warn the user of potential dangers

To prevent the program from taking undesirable actions, the admonition approach would be to show a warning so that the user has an opportunity to intervene before the undesirable action happens. Commercial products like ZoneAlarm (see Chapter 27) do just this, stepping in with a prompt when programs access the network. Unfortunately, the computer can't read the user's mind, so it has to guess when to warn and when to proceed. Warning too little places the user at risk; warning too much annoys the user. The larger the discrepancy between acceptable and allowed actions, the more severely we are forced to compromise between security and usability.

Security by designation

Figure 13-7 acknowledges that user expectations change over time and that the security policy should change to match. In this approach, the program starts out with minimal authority (solid box). As before, the user can interact with the program's user interface (solid arrow) to have the program carry out actions using its initial authority. When the user wants to take an action requiring new authority, the user interacts with the system's user interface (dotted arrows) to express both the command and the extension of authority (dotted box) at the same time. Combining the authorization with the designation of intent in a single user action maintains a close, dynamic match between the allowed set and the acceptable set of actions. Users don't have to establish a detailed security policy beforehand and don't have to express their intentions twice.

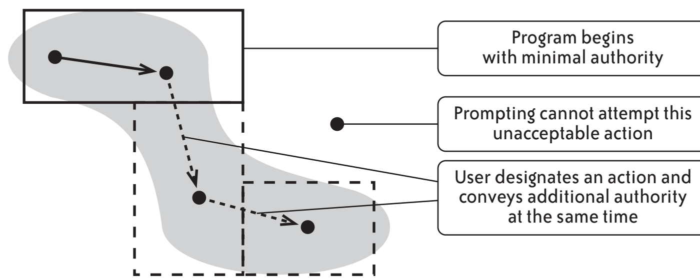


FIGURE 13-7. With security by designation, the user simultaneously designates an action and conveys the authority to take that action

Advantages of designation

Software commonly employs a mix of these two approaches. Launching a program is a case of security by designation: users don't have to select a program and then separately grant memory to the newly created process; a single action accomplishes both. Sending an email attachment is another example of designation: users don't have to select a file to attach and then separately assign read permissions to the recipient of the message; dropping the attachment into the message designates the attachment, the recipient, and the intended authorization in a single act. When a web browser asks for confirmation before submitting a form, or when Microsoft Word asks whether to enable macros upon opening a document, these are cases of security by admonition.

When security by designation is possible, it is convenient and straightforward because it achieves the ideal of integrating security decisions with the user's primary task. On the other hand, security by admonition demands the user's attention to a secondary source of information. With designation, the authorization is part of a user-initiated action, so the user has the necessary context to know the reason for the authorization. With admonition, the request for authority is initiated outside the user, so the user may not have the context needed to decide whether to approve it.

Implementing security by designation

To implement the designation strategy, we have to find an act of designation with which to associate the authorization. When we are tempted to ask the user whether a particular action is acceptable, we instead determine how the user originally specified that action. The action may have been conveyed through several software components, so we must follow it back to the originating user interaction. Then we move that user interaction into a software layer that the user trusts to handle the relevant authority and convey the authority together with the designation of the action. The example in the later "Securing file access" section will demonstrate how this is done.

Implementing security by admonition

It may be necessary to fall back on admonition when security by designation isn't feasible—for example, if the original designating interaction is inaccessible or untrustworthy. It is usually better to inform users without interrupting their workflow. Forcing users to answer a prompt is a terrible way to present a notification: it teaches users that security issues obstruct their work and trains them to dismiss prompts carelessly instead of making meaningful decisions.

Seek designs that are noticeable by their proximity and relevance to the matter at hand, not by their aggressiveness. For example, the Firefox web browser notifies the user when an unknown site tries to install software by adding a transient bar to the browser window; the bar does not prevent the user from viewing the page as a prompt box would. Notifications concerning a particular operation can be displayed just for the duration of the operation. Later in this chapter, Figure 13-15 shows a note about passwords appearing next to a password field while the field is active; the note disappears when the user moves to another field. Displaying tips near the mouse cursor or changing the mouse cursor are also ways to provide clear yet noninterrupting feedback.

User-Assigned Identifiers

Practically every user interaction depends on the correct designation of an object or action. The names by which a user refers to objects and actions come from many different sources: some are assigned by the user, some are assigned by other users, some are allocated by global or local organizations, and some are assigned by programs.

Giving objects the power to choose their own names places the user at a significant disadvantage. Unless another naming mechanism is provided, the user is forced to work with identifiers controlled by outside parties—potentially, to communicate with his most trusted software agents using terms defined by an adversary. Each name that enters the user's sphere is a potential avenue of attack if it can be made misleading or confusing.

A good way to avoid the danger of identification problems is to let users assign and use their own local identifiers for objects and actions. Computers are harder to confuse than humans; they can take care of the translation between users' familiar names and the names used by machines or other people.

You already use user-assigned identifiers all the time. For example, each file on your desktop has a name that you can assign. The true, unique identifier for each file is a number that tells the system how to locate the file on the disk. But you never have to deal with that number; you assign a filename, and the operating system takes care of translating between the name and the number. Imagine how difficult it would be to identify files if the desktop displayed only disk addresses instead of filenames, or how inconvenient it would be if files all chose names for themselves that you couldn't change.

Another means of user-controlled identification is the assignment of positions to icons. Preserving the positions of the icons on the desktop helps you find them later. Some systems also let you assign colors to icons. These mechanisms give you more control over how you identify objects, improving your ability to designate them accurately.

Applying the Strategies to Everyday Security Problems

Let's look at a few examples to see how security by designation and user-assigned identifiers help us address security problems in practice.

Email viruses

Self-propagating email attachments have caused widespread havoc in the last few years. Some exploit software bugs in the operating system or mail program, but bugs are not the whole story. Many email viruses, such as MyDoom, Netsky, and Sobig, rely on humans to activate them by opening executable attachments, and would continue to spread even with bug-free software.

On Microsoft Windows and Mac OS systems, double-clicking serves two very different purposes: opening documents and launching applications. Whereas documents are usually inert, starting an application grants it complete access to the user's account. Thus, a user can double-click an attachment, intending only to read it, and instead hand over control of the computer to an email virus.

The missing specification of intent here is the choice between *viewing* and *executing*. The user is not given a way to make this choice, so the computer guesses, with potentially dangerous consequences. A bad solution would be to patch over this guess with a prompt asking the user "Do you really want to run this application?" every time an application icon is double-clicked. To find a better solution, we need to consider how users specify the intent to run a new application.

What do users already do when they install applications? On a Mac, most applications are installed by dropping a single icon into the Applications folder. So, suppose that double-clicking files in the Applications folder launches them, and double-clicking files elsewhere only passively views them. Suppose also that the Applications folder can only contain files that the user has placed there. The resulting system would respect a distinction between viewing and executing established by the convention users already know. Programs would run only if installed by the user. The main propagation method of email attachment viruses would be eliminated with virtually no loss of usability, because Mac

users normally run their applications from the Applications folder. As for Windows, dropping a single icon is simpler to do and understand than running a multistep application installer, so switching to a drag-and-drop style of installation would simultaneously improve *both* security and usability for Windows users.

Other viruses and spyware

Although distinguishing viewing from execution would be better than what we have now, it provides a defense only when opening email attachments. Any program the user installs voluntarily, including downloaded programs that might contain viruses or spyware, would still run with the user's full access rights, exposing the user to tremendous risk.

The lack of fine-grained access controls on application programs is an architectural failure of Windows, Mac OS, and Unix systems. Such fine-grained control would enable a true solution to the virus and spyware problems. Let's consider how to control two major kinds of access exploited by viruses: file access and email access.

Securing file access

How do users specify the intent to read or write a particular file? In current graphical user interfaces, files are mainly designated in two ways: by pointing at file icons and by selecting filenames in dialog boxes. With Mac OS and Windows, both of these functions are implemented in the operating system. However, selecting a file passes the file's name to an application, not the file itself. The application then asks the operating system to open the file by name, receiving an object called a *filehandle* that represents the file. Applications on Windows, Mac OS, and Unix systems assume the user's identity when accessing the disk, so they can ask the system to open any of the user's files.

Here's an analogy to illustrate the concept of filehandles. Suppose that I have a telephone and I'd like to call my mother to wish her a happy birthday, but unfortunately I am a terrible singer. On the other hand, you have no telephone, but you have a wonderful voice and kindly offer to sing "Happy Birthday." I have two options: I could let you use my telephone, give you my mother's phone number, and ask you to call her (Figure 13-8a). But that would let you dial any number and call anyone with my phone, just as an application can "dial up" any file by its name. Instead, I could call my mother on the phone and then hand you only the receiver (Figure 13-8b). That way, I'd be sure that you were speaking only to her, and I wouldn't have to reveal her phone number to you. I could even disconnect you if necessary. Handing you a telephone receiver that's connected to another party is like giving a filehandle to an application.

To perform security by designation, we would stop providing applications with all the user's authority to access the disk; applications would start with access only to their own program files and a limited scratch space. Instead of returning a filename, the file dialog would open the selected file and return a filehandle to it (Figure 13-9). Similarly, dropping a file on an application would send the application a filehandle rather than a

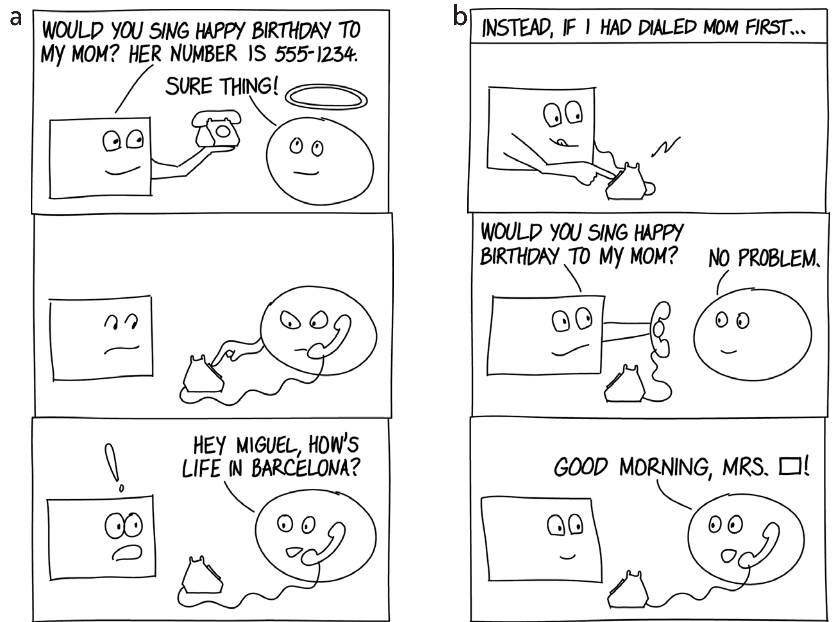


FIGURE 13-8. (a, left) Handing over the entire telephone conveys the authority to dial any number; (b, right) handing over only the receiver is more consistent with the principle of least authority

filename. The user experience of opening files would be exactly the same, yet security would be improved significantly because programs would get access only to user-selected files. Viruses and spyware wouldn't be able to install themselves or modify files without the user's specific consent.

Some special programs, such as search tools and disk repair utilities, do require access to the entire disk. For these programs, the user could convey access to the entire disk by installing them in a "Disk Tools" subfolder of the Applications folder. This would require a bit more effort than dropping everything in the Applications folder, but it isn't an alien concept: on Windows systems, for example, disk utilities are already kept in an "Administrative Tools" subfolder of the Start Menu.

Securing email access

How do users specify the intent to send email to a particular person? They usually choose recipients by selecting names from an address book or by typing in email addresses. To control access to email capabilities, we might add a *mailhandle* abstraction to the operating system, which represents the ability to send mail to a particular address just as a filehandle represents the ability to read or write a particular file. Then, selecting names from a standard system-wide address book would return mailhandles to the application, just as a secure file selection dialog would return filehandles. Using mailhandles would allow us to stop providing general network access to all programs, rendering viruses unable to email themselves to new victims.

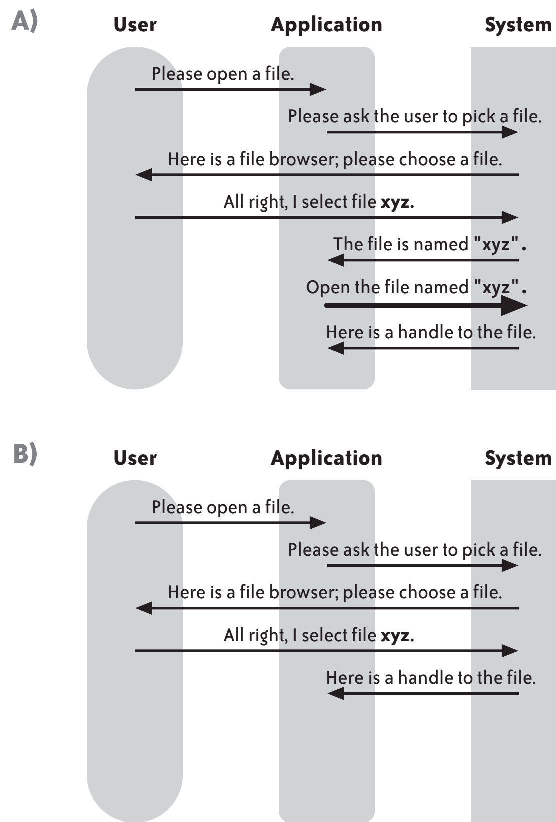


FIGURE 13-9. (a, top) In today's Mac OS and Microsoft Windows systems, file dialogs return a filename; applications use their unlimited user file access (bold arrow) to open the selected file; (b, bottom) returning a filehandle is simpler and more secure while requiring no visible changes to the user interface; applications' access to user files is restricted to only files that the user designates, thereby protecting the user from malicious programs such as viruses and spyware

Cookie management

Cookies are small data records that web sites ask browsers to retain and present in order to identify users when they return to the same site. They enable web sites to perform automatic login and personalization. However, they also raise privacy concerns about the tracking of user behavior and raise security risks by providing an avenue for circumventing logins. Users need some control over when and where cookies are sent.

Many browsers address this issue by prompting users to accept or reject each received cookie. But cookies are so ubiquitous that this causes users to be constantly pestered with irritating prompt boxes. To reduce workload, some browsers provide an option to accept or reject all cookies from the current site, as in Figure 13-10. After one has chosen a site-wide policy, it can be tricky to find and reverse the decision. Yet the decision to *accept* cookies is immaterial, as the real security risks lie in *sending* cookies. Moreover, the term "cookie" refers to an implementation mechanism and has no relevance to user tasks; if cookies appear at all in the interface, their appearance should relate to the function they serve.

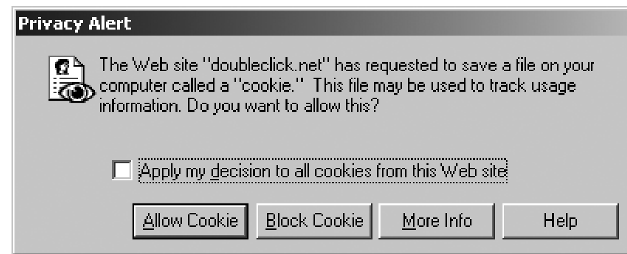


FIGURE 13-10. Internet Explorer 6.0 prompts the user to accept a cookie

The missing specification of intent here is whether the user, upon returning to a web site, wants to continue with the same settings and context from the last session. To find a better solution, consider how users specify that they want to return to web sites they've seen before.

The existing user interface mechanism for this purpose is the bookmark list: users designate the site they want by selecting a bookmark. Therefore, suppose that each bookmark has its own associated "cookie jar." One jar at a time would be the active cookie jar, from which cookies are sent and where received cookies are stored. When the user creates a bookmark, the cookies for the current session are placed in the new bookmark's cookie jar,[9] and it becomes the active cookie jar. When the user selects a bookmark, its cookie jar becomes active. This is a change from today's typical browsers, but the usage is easily explained: "Bookmark a site if you want to continue later where you left off."

Bookmarks containing cookies would be displayed with a mark to show that they are personalized. In the example in Figure 13-11, personalized bookmarks are marked with a small heart symbol. A bookmark icon in the location bar would indicate whether there are bookmarks for the current site, and it would also be marked to show whether the current view is personalized. Clicking on the icon would show a menu of bookmarks for the site as in Figure 13-12, enabling users to switch to a personalized session if they want.

This solution would eliminate the need for prompts or lists of cookie-enabled and cookie-disabled sites to manage. In fact, there would be no need to mention "cookies" in the user interface at all; one would simply describe some sessions as personalized. Login cookies would no longer be left lingering on public-access machines. Users' privacy would be better protected. Sites would no longer mysteriously change their appearance depending on hidden state. And users would gain additional functionality: advanced users could manage multiple logins or multiple suspended transactions simply by creating bookmarks. Security and usability would both be simultaneously improved.

9 Some cookies are temporary, marked to expire automatically at the end of a session. Others are persistent, marked to be saved for future sessions. In the design proposed here, only persistent cookies issued by the bookmarked site would be stored in the cookie jar with a bookmark. Cookies issued by third parties would be ignored.

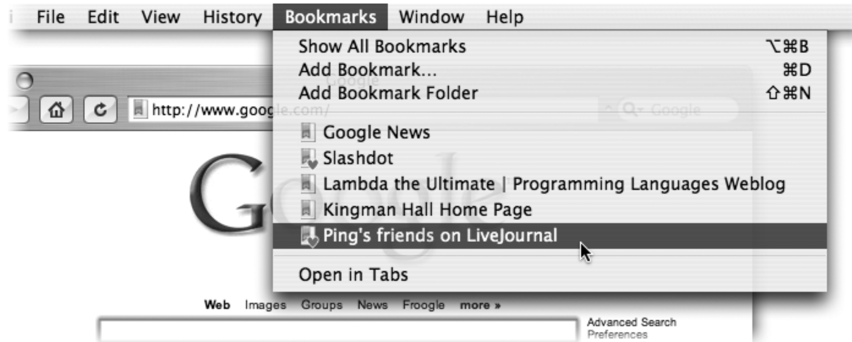


FIGURE 13-11. In the proposed design, bookmarks that the user has created for personalized sessions are marked with a small heart in the bookmark list; selecting such a bookmark restores the session

FIGURE 13-12. (a, left) A bookmark icon next to the URL indicates that the user has bookmarks for the current site; the user can select a bookmark to switch to a personalized session; (b, right) an advanced user has created and named bookmarks for two different logins; the bookmark icon next to the URL is marked with a small heart to show that the current session is personalized, and the drop-down menu allows the user to switch to a different personalized session

Notice that security by designation has helped us to arrive at a design that follows several of the guidelines identified earlier: by requiring a user action to save cookies instead of a user action to reject cookies, we match the safer option with the path of least effort. By activating cookies based on the selection of a bookmark, we associate the sending of cookies with an authorizing act. By identifying which session is active, we help the user maintain awareness of the authority the user wields. And instead of having to express policy decisions in the unfamiliar language of cookies, the user expresses these decisions by creating and using bookmarks, which serve a useful and familiar task.

Phishing attacks

Forged email messages and web sites designed to steal passwords are a common method of stealing private information. SMTP, the dominant Internet mail protocol, places no restrictions on the "From:" field of an email message, so messages can be easily forged to appear to come from anyone. In a typical so-called *phishing attack*, a user receives an email message that appears to be from a bank, asking the user to click on a link and verify account information. The link appears to point to the bank's web site, but actually takes the user to an identity thief's web site, made to look just like the bank's official site. If an unsuspecting user enters personal information there, the identity thief captures it.

Here, the missing specification of intent is the intended recipient of the form submission. In one scam, for example, an imitation of PayPal was hosted at *paypai.com*. This problem is tricky because the system has no way of knowing whether the user intended to go to *paypal.com* and was misdirected to *paypai.com*, or whether the user really wanted to go to *paypai.com*. The intention resides only in the user's mind.

The designation at the heart of this security problem is the link address. Yet the specification of that address does not come from the user; the user merely clicks in the email message. Our problem would be solved if we could secure the email channel and determine the source of the message. But until we can convince most users to switch to secure email, there is no trustworthy designation on which to hang a security decision.

In this case, practical constraints make security by designation infeasible. The user is required to identify a web site by observation rather than by designation. The site has total control over its appearance; even the URL is mostly chosen by the site itself.

To protect the user, we have to provide some helpful information. Traditional admonition-style thinking would suggest warning the user when submitting forms to a suspicious-looking site. Such a prompt might look something like Figure 13-13.

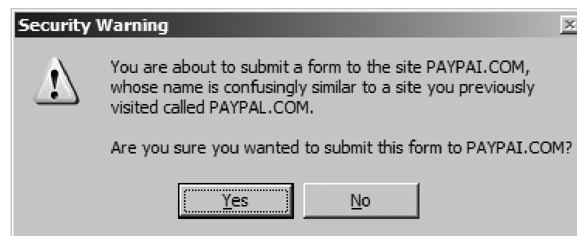


FIGURE 13-13. Prompting is a traditional method of phishing protection

Indeed, this is exactly the approach taken by many antiphishing tools. Perhaps the most sophisticated is a commercial program called Web Caller-ID, which employs "hundreds of routines that examine the elements of a web site" to decide whether a site is fraudulent.[10] This type of solution is unreliable because it makes guesses based on assumptions about good and bad sites. The creators of Web Caller-ID claim that their routines can detect 98% of spoof sites—an impressive figure. But because there are more than 1,400 different phishing attacks per month (and this number is rising),[11] and each one could target thousands of users, the missed 2% can still do considerable damage. Moreover, any scheme based on heuristics generates spurious warnings. A false-positive rate of even 1% could be enough to train users to ignore warning messages.

10 WholeSecurity, "Web Caller-ID Core Technology"; *http://wholesecurity.com/products/wcid_core_technology.html*.

11 Anti-Phishing Working Group, "Phishing Attack Trends Report, June 2004"; *http://antiphishing.org/APWG_Phishing_Attack_Report-Jun2004.pdf*.

A better solution is to give the user control over the name used to identify the site. The petname toolbar[12] provides a text field, as shown in Figure 13-14, where the user can enter a *petname* for the current site (a nickname for the site specific to just this user). The user would assign a name to a site when registering for an account at the site.

FIGURE 13-14. The petname toolbar, shown in untrusted (top) and trusted (bottom) states, supports user-assigned site names

The user-assigned name appears in the text field whenever the user returns to the same site. If the user hasn't assigned a name, the field shows that the site is unknown. As long as the user is aware of the petname, it doesn't matter what domain name a scammer uses. The act of assigning the name is initiated by the user, at the time the user chooses to enter a trust relationship with the site, in a context that the user understands.

This way of browsing would be completely safe against misidentification, without any heuristics or false positives, as long as the user notices the petname in the toolbar. Checking the toolbar is a change from current practice, so an admonition that brings the petname closer to the user's natural workflow might be needed to help users migrate. Figure 13-15 shows a message that might appear as the user enters a password on a site he has not named. The admonition is a temporary message, not a prompt window, so it informs the user without interrupting. Interrupting the user with a prompt would defeat our purpose: it would encourage users to assign petnames to sites in response to the prompt, instead of assigning names on their own initiative.

This petname scheme contrasts with today's centralized public key infrastructure (PKI) of digital certificates issued by certification authorities. PKI was intended to prevent just the type of misidentification that is exploited in phishing attacks, yet phishing is rampant. Some might say the problem is that users don't check certificates. But users rightly feel no compulsion to rely on a naming system with which they have no involvement and no trust relationship. (I'll bet you don't know who issued the certificate for the last secure

12 Waterken Inc., "Petname Toolbar"; *http://www.waterken.com/user/PetnameTool/*.

FIGURE 13-15. An admonition appears while the user is entering a password into a scam site

web site you visited.) Carl Ellison has identified many problems with centralized PKI.[13,14] Chapter 16 describes how key management can be made vastly simpler and more useful by using local namespaces instead of centralized authorities. Enabling users to assign personally meaningful names frees users from external sources of confusion, political battles, or dependence on unfamiliar third parties. See the Pet Name Markup Language[15] for further elaboration of the petname concept.

Real implementations

CapDesk[16] and Polaris[17] are two research projects that apply the design ideas and strategies described in this chapter:

- *CapDesk*. A desktop shell that lets users safely run downloaded software in a familiar graphical environment. The CapDesk system and applications are written in the secure programming language, E.[18]

13 Carl Ellison and Bruce Schneier, "Ten Risks of PKI: What You're Not Being Told About Public Key Infrastructure," *Computer Security Journal* 16 (Winter 2000); *http://www.schneier.com/paper-pki.pdf*.

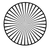
14 Carl Ellison, "Improvements on Conventional PKI Wisdom," *Proceedings of the 1st Annual PKI Research Workshop* (NIST, 2002); *http://www.cs.dartmouth.edu/~pki02/Ellison/*.

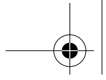
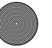
15 Mark Miller, "Lambda for Humans: The Pet Name Markup Language"; *http://erights.org/elib/capability/pnml.html*.

16 David Wagner and Dean Tribble, "A Security Analysis of the Combex DarpaBrowser Architecture"; *http://combex.com/papers/darpa-review/index.html*.

17 Marc Stiegler, Alan Karp, Ka-Ping Yee, and Mark S. Miller, "Polaris: Virus Safe Computing for Windows," HP Labs Technical Report HPL-2004-221; *http://www.hpl.hp.com/techreports/2004/HPL-2004-221.html*.

18 Mark Miller, "E: Open Source Distributed Capabilities"; *http://erights.org/*.



- *Polaris.* A safe environment for running existing Microsoft Windows applications. With Polaris, one can use programs normally, enable macros, and open email attachments without experiencing ill effects from viruses or spyware.

Both of these projects are based on the *object-capability paradigm*,[19] in which combining designation with authority is a central design principle. The object-capability model has a long history of development in computer systems research, particularly secure operating systems[20,21] and secure programming languages.[22] It is no coincidence that object-capability design yields benefits at the user level as well as the system level: the object-capability model has its roots in object-oriented programming, which was a breakthrough at both levels.

Conclusion

This chapter has offered advice at two levels: the guidelines describe qualities that usable secure software should have, and the strategies describe ways to design software with those qualities. Although these strategies address only some of the guidelines and are far from a complete solution, they have been effective in many situations. As we saw in our discussion of how phishing exploits email's weaknesses, integrating with insecure installed systems yields some of the toughest design problems and can prevent the direct application of these strategies. Even in such cases, the guidelines and strategies can help highlight vulnerable areas of a system and improve the design of countermeasures.

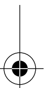
The theme of user initiation links the two strategies presented here. With security by designation, the user proactively designates an action instead of reacting to a notification that appears out of context. With user-assigned identifiers, the user proactively assigns names instead of reacting to names that are controlled by another party. Security based on actions initiated by the user more accurately captures the user's intentions than security based on the user's response to external stimuli. When the user initiates, security works for the user instead of the user working for security.

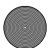
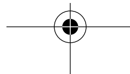
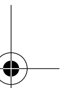
19 Miller and Shapiro.

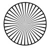
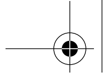
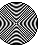
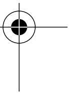
20 Norman Hardy, "The KeyKOS Architecture," *Operating Systems Review* 19 (October 1985), 8–25; *http://www.cis.upenn.edu/~KeyKOS/OSRpaper.html*.

21 Jonathan Shapiro, "EROS: A Fast Capability System," *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (New York: ACM Press, 1999); *http://www.eros-os.org/papers/sosp99-eros-preprint.ps*.

22 Miller, "E: Open Source Distributed Capabilities."



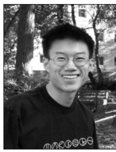




Acknowledgments

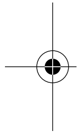
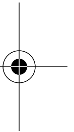
The 10 guidelines in this chapter are based on an earlier set of principles[23] developed in collaboration with Miriam Walker through a series of discussions with Norm Hardy, Mark S. Miller, Chip Morningstar, Kragen Sitaker, Marc Stiegler, and Dean Tribble. Thanks to Morgan Ames, Tyler Close, Lorrie Cranor, Paul Eykamp, Simson Garfinkel, Jeremy Goecks, Alan Karp, Linley Erin Hall, Marti Hearst, Siren Torsvik Henriksen, Rebecca Middleton, Mark S. Miller, Diana Smetters, and Marc Stiegler for their many helpful suggestions, and to David T. Wallace for his advice on illustrations.

About the Author



Ka-Ping Yee is a Ph.D. student at the University of California, Berkeley. His research interests include interaction design, capability-based security, information visualization, and computer-supported argumentation.

http://zesty.ca/



23 Ka-Ping Yee, "User Interaction Design for Secure Systems," in Robert Deng, Sihan Qing, Feng Bao, and Jianying Zhou (eds.), *Proceedings of the 4th International Conference on Information and Communications Security, Lecture Notes in Computer Science 2513*, (Heidelberg: Springer-Verlag, 2002); *http://zesty.ca/sid/*.