Aligning Security and Usability

Conflicts between security and usability goals can be avoided by considering the goals together throughout an iterative design process. A successful design involves addressing users' expectations and inferring authorization based on their acts of designation.

KA-PING YEE
University of California, Berkeley

Designers of security-sensitive software applications sometimes speak of a trade-off between achieving strong security and making software easy to use. When we look for ways to adjust an existing design, usability improvements seem to yield more easily compromised software, and adding security measures seems to make software tedious to use or hard to understand. Yet designers cannot afford to neglect either—both security and usability failures can render a product useless.

Conflicts between security and usability can often be avoided by taking a different approach to security in the design process and the design itself. Every design problem involves trading off many factors, but the most successful designs find ways to achieve multiple goals simultaneously. To this end, in this article I discuss when and how we can bring security and usability into alignment through these main points:

- Security and usability elements can't be sprinkled on a product like magic pixie dust. We must incorporate both goals throughout the design process. When security and usability are no longer treated as add-ons, we can design them together to avoid conflicts.
- We can view security and usability as aspects of a common goal: fulfilling user expectations. This involves maintaining agreement between a system's security state and the user's mental model, both of which change over time.
- An essential technique for aligning security and usability is incorporating security decisions into the users' workflow by inferring authorization from acts of designation that are already part of their primary task.

I apply these points to three classes of everyday security problems: worms, cookie management, and phishing attacks.

Scope

Two assumptions frame my discussion. First, I assume a known user, setting aside the problem of authenticating the user. This article will not contribute any insights on authentication.

Second, I assume that the parties we intend to serve have a mutually understood framework of acceptable behavior. (Malicious parties might attempt unacceptable behavior, but we are not interested in serving their needs.) For example, if music distributors wish to impose copying restrictions that music listeners find unreasonable, then software designers cannot serve both distributors and listeners without compromising. In such a situation, the conflict stems not from usability issues but from disagreements between people about policy, which is a different problem. Resolving such disputes is outside the scope of this discussion.

Conflicts in the design process

To understand how security and usability come into conflict during software design, it might be helpful to look at this tension from both points of view.

Pseudosecurity harms usability

As a security practitioner, you might have been asked to take a nearly complete product and make it more secure. You then know firsthand how difficult and ineffective it is to try to add on security at the last minute. Although we

can find some types of bugs in a code review, real security is a deeper property of the whole design. John Viega and Gary McGraw wrote, "Bolting security onto an existing system is simply a bad idea. Security is not a feature you can add to a system at any time."[1]

What happens when people try to bolt on security instead of designing it into a system from the ground up? Among other things, usability suffers. After-the-fact security fixes often add configuration settings and prompts. But extra settings and prompts aren't very effective at solving security problems—they just make it easier to blame user error when something goes wrong.

Pseudousability harms security

If you're a usability practitioner, you might have had to take a nearly complete product and make it more usable. Don Norman described this mistake: "[The] assumption that user experience is just another add-on is pretty consistent across the industry."[2] Thus, you might know how difficult and ineffective it is to try to add on usability late in the development process. Good usability engineering requires understanding user needs and incorporating the appropriate features throughout the design process, not tacking on superficial features such as flashy widgets, animations, or skins.

What happens when people try to bolt on usability instead of designing it into a system from the ground up? Among other things, security suffers. Usability quick fixes sometimes involve hiding security-related decisions from the user or choosing lax default settings. Flashy, nonstandard interfaces can also decrease a product's security by increasing complexity or confusing users.

Integrated iterative design

Both the security and usability communities have advocated iterative development processes based on repeated analysis, design, and evaluation cycles, rather than linear processes in which security or usability testing occurs at the end. Although many teams have adopted iterative processes, few seem to incorporate security and usability throughout. Not only is it important to examine these issues early and often, it is vital to design the user interface and security measures together. Iterating offers the opportunity to see how security and usability decisions affect each other—in fact, separating security engineering from usability engineering virtually guarantees that conflicts will arise.

Conflicts during use

At first glance, the source of conflict might appear obvious: security usually aims to make operations harder to do, while usability aims to make operations easier. However, it's more precise to say that security restricts access to operations that have undesirable results, whereas usability improves access to operations that have desirable results. Users designate which results are desirable with their actions in the user interface. When a system accurately interprets these actions, security and usability do not conflict, and the problem becomes merely a matter of functional correctness. Thus, conflicts between the two goals arise when a system lacks the information to determine whether a particular result is desirable.

Presenting security to users as a secondary task promotes conflict. People typically use computers for purposes other than security, such as communicating with friends, using online services, managing time and money, composing and editing creative work, and so on. Asking users to take extra security steps is problematic because the extra steps either interrupt their workflow or end up hidden in configuration panels. Interrupting users with prompts presents security decisions in a terrible context: it teaches users that security issues obstruct their main task and trains them to dismiss prompts quickly and carelessly.

Both arguments suggest that we can help bring security and usability into alignment by seeking ways to extract and use as much accurate information as possible from a user's normal interactions with the interface. As the following examples will illustrate, the tediousness of security features sometimes comes from throwing away that information. The more information about security expectations we can determine from the actions that the user makes naturally in carrying out a primary task, the less we need to insert secondary security tasks.

Mental models

Although software components are often casually labeled *trusted*, trustworthiness is not a black-and-white issue. The word *trusted* is meaningless without answers to the questions, "Trusted by whom?" and "Trusted to do what?" We cannot define security policies without asking, "Secure from whom?" and "Secure against what?"[3] Attempting to classify all programs as simply trusted or untrusted is not helpful, yet some security experts continue to think along such lines.[4] The missing component in this common and dangerous oversimplification is the *mental model*.

Simson Garfinkel and Gene Spafford wrote that "a computer is secure if you can depend on it and its software to behave as you expect."[5] Fulfilling expectations is a matter of keeping behavior and expectations in agreement, and the users' expectations are based on their mental model of the system. Both the security policy and mental model are dynamic; they change in response to user actions.

The basic question of *who* can do *what* suggests a framework for mental models based on *actors* and *abilities*.[6] Actors are entities that the user perceives as independent agents—for example, application programs or other users. At any given moment, each actor possesses a set of abilities—potential actions that can affect the user. (Ac-

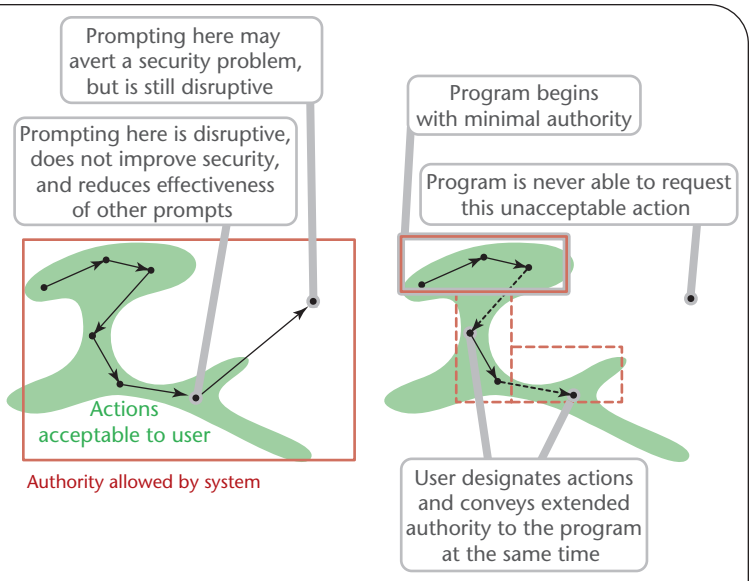

Figure 1. Space of possible actions. (a) Security by admonition: The red rectangle delineates the actions that *B* can request through *A*. The green curvy region shows the set of actions acceptable to the user in a particular situation. The black dots represent the actions that *B* exercises. The solid arrows are the user's interactions with *B*'s user interface that motivate *B* to take those actions; (b) security by designation: the dotted arrows are interactions with *A*'s user interface that simultaneously grant additional authority to *B* while conveying the user's intended command to *B*.

tors and abilities are analogous to principals and capabilities but are parts of the user's mental model, not the implementation.) The actor–ability state in the user's mental model should accurately bound the actual state of access rights in the system at all times. To use this framework, we perform usability studies to determine which actors and abilities are present in the mental model and how users expect their interactions with the computer to change the actor–ability state.

Admonition and designation

Consider a software component *A* that a user trusts to constrain the effects of another component *B*, which affects the user's world by making requests through *A*. For example, *A* might be an operating system and *B* might be an application running on it, or *A* might be a Web browser and *B* might be a script or plug-in on a Web page.

Figure 1 shows abstract diagrams of the space of possible actions. In Figure 1a, the red box is a static, pre-established security policy intended to limit *B* to a set of safe actions in the vicinity of the green region, which represents the actions the user considers acceptable. Because the policy is only finitely detailed, it only roughly approximates the exact set of abilities that the user wants *B* to have. Because the policy is static, it must also accommodate the various ways that we might want to use *B* in other situations. Thus, the red box includes large areas not in the green region. (Windows and Unix systems are extreme examples; applications run with the user's full privilege, so an enormous discrepancy exists between the allowed set and the acceptable set of abilities.)

Security by admonition

What happens when *B* attempts an action unacceptable to the user? One strategy is to help the user avoid letting *B* exercise the action by explaining what will happen or intervening with a request for confirmation. This approach is *security by admonition*. Unfortunately, the computer doesn't know exactly what the user considers acceptable, so it can't know when to warn. It can guess what most users would consider acceptable and trade off the inconvenience of prompting against the magnitude of potential damage. Nonetheless, there are bound to be false positives and negatives. Bad prompts make users wonder, "Why is the computer asking if I want this? I already told it to do this." The larger the discrepancy between attemptable and acceptable actions, the more severely we are forced to compromise between security and usability. In other words, violating the principle of least privilege[7] forces security and usability into conflict.

Security by designation

Figure 1b acknowledges that the security policy and user expectations change over time and depicts a different strategy: *security by designation*. In this approach, *B* starts with a minimal set of abilities. When the user wants *B* to do various things, the user's actions simultaneously express the command and the extension of authority. As before, the solid arrows are interactions with *B*'s user interface. The dotted arrows are interactions with *A*'s user interface that simultaneously grant additional authority to *B* while conveying the user's intended command to *B*. We thereby maintain a close match between the allowed set and the acceptable set of abilities without asking users to establish a detailed security policy beforehand or express their intentions twice.

Software systems employ a mix of these two strategies. For instance, launching an application is a case of designation; users don't need to select an application and then separately grant CPU time to the newly created process because a single action accomplishes both. Sending an email attachment is another example; users don't have to select a file to attach and then separately grant read permissions to the message's recipient because attaching the file conveys both designation and authorization. Security warnings and prompts, including any questions beginning with, "Are you sure...?," or, "Do you want to allow...?," are examples of admonition.

Security by designation is convenient and straightforward because it embodies the aforementioned ideal of integrating security decisions with the user's primary task. Security by admonition, on the other hand, demands the user's attention to a secondary source of information. With designation, the user grants authority, so the user has the necessary context to know why it should be granted. With admonition, the program initiates the request for authority, so the user might not have the context to decide whether to grant it. Therefore, we should use security by designation whenever feasible.

Implementation

To implement the designation strategy, we have to find the act of designation on which to hang the security decision. When tempted to ask the user, "Is action *X* acceptable?," we instead ask ourselves "How was action *X* specified?" The action might have been conveyed through various software components, so we must follow it back to its origin. When the origin is a user interface interaction, we lift that interaction into a software layer that the user trusts to handle the relevant authority (for example, from *B* into *A*) and convey the authority together with the designation of the action.

However, security by designation might not be feasible for various reasons. The authority-granting action might be inaccessible, interoperating with other systems might make the designation of actions untrustworthy, or the effort required to support finer-grained control might be too great. In these cases, we could be forced to fall back on security by admonition.

Admonition is required whenever the user is likely to grant an actor the ability to do something that the user doesn't want. This might happen due to user error, because the harmful action might not be technically preventable, or because the security primitives are too coarse to distinguish what the user wants. When using admonition, be careful about leaping from stating factual consequences to passing judgment on whether an action is good or bad. Information about an action's outcome can be displayed neutrally without interrupting the user's workflow. Intervening with a warning prompt is disruptive and likely to cause frustration. For example, opening any attachment causes Microsoft Outlook to display a warning about the possibility of viruses. The warning offers two choices: to open the file now or save it to disk, whereupon presumably the user will go and open it anyway. Issuing such warnings too frequently reduces the user's trust in admonitions, decreasing their effectiveness in critical situations. Thus, admonition should strive to inform, not interrupt.

Admonition and designation are not mutually exclusive strategies, but admonition should be avoided when security by designation is acceptable.

Design problems

To make these ideas more concrete, let's apply them to some security problems that computer users commonly experience. We will examine how to adjust a design to implement security by designation and how to know when it's necessary to fall back to security by admonition.

Viruses and worms

Self-propagating email attachments have caused widespread havoc in the past few years. Some of them exploit software bugs in email clients. Fixing software bugs is an important step toward eradicating these worms, but bugs are not the whole story. Many email worms, such as MyDoom, Netsky, and Sobig, rely on humans to activate them by opening executable attachments; they would continue to run rampant even with bug-free software.

Viewing and executing. On Windows and Mac OS machines, the act of double-clicking opens documents and launches applications. Documents are usually inert, whereas launching an application grants it complete access to a user's account. Thus, by double-clicking on an attachment, users who intend only to read it instead hand over control of their computer to a nasty worm.

A missing piece of information here is the choice between *viewing* and *executing*. The user is given no way to communicate this choice, so the computer has to guess, with potentially dangerous consequences. A poor solution would be to patch this guess by asking the user, "Do you really want to start this application?," every time he or she double-click on an application icon. To find a better solution, we must consider how users specify their intent to run a new application.

Let's look at what users already do when they install applications. On a Mac, users install most applications by dropping them into the Applications folder. Suppose that double-clicking icons in the Applications folder launched them, but double-clicking icons elsewhere would only passively view them. This would stop the main propagation method of worms attached to emails with virtually no loss of usability—Mac users usually run applications from the Applications folder anyway. As for Windows users, dragging a single icon is simpler than running a multistep application installer, so switching to a drag-and-drop style of installation would simultaneously improve security and usability.

Although distinguishing viewing from execution would be better than what we have now, this distinction is rather blunt. The suggested solution is not much good for threats other than executable attachments. Any program the user actually installs, including downloaded software that could contain viruses or spyware, would still run with full user-level access, exposing the user to tremendous risk.

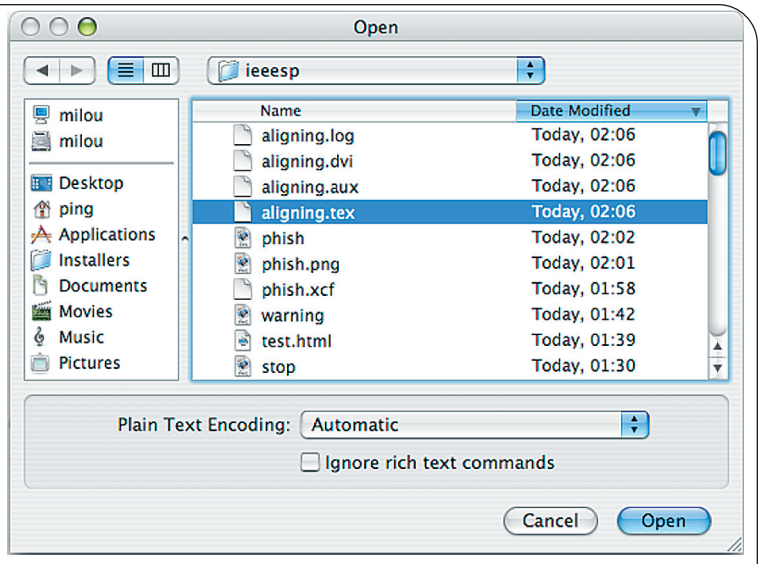

Figure 2. A file selection dialog box. We can achieve much stronger file access protection against viruses and Trojan horses without any change to this user interface.

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

How do users specify the intent to read or write a particular file? In current GUIs, files are designated by pointing at file icons and selecting file names in dialog boxes. Standard functions in Mac OS and Windows already implement both of these interactions. To perform security by designation, we would stop providing applications with default access to the file system and instead provide them with file access via icon manipulation and the file selection dialog box. Applications would start with access limited to their own program files and a bounded scratch space. Instead of receiving a file name from the file dialog box and then using universal file system privilege to open the associated file by name, an application would receive a file handle directly from the dialog box. Similarly, dropping a file on an application would send the application a message containing the file handle instead of the file name. The user experience would not change, yet security would be vastly improved.

Certain special programs, such as search tools or disk utilities, do require access to the entire disk, and it would be impractical to designate files individually. Instead, the user could convey access to the entire disk by installing programs 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 is not an alien concept—on Windows systems, disk utilities are already kept in an Administrative Tools subfolder of the Start menu.

How do users specify the intent to send email to a particular person? Users usually indicate recipients by selecting names from an address book or typing in email addresses. To control access to email, we might add a *mail handle* abstraction to the operating system, which represents the ability to send mail to a particular address just as a file handle represents the ability to read or write a particular file. Then, selecting names from a standard system-wide address book would return mail handles to the application, just as a secure file selection dialog box would return file handles. Mail handles would let us stop providing default network access to all programs, so that worms could not email themselves to new victims. To allow a mail client to implement its own custom address book and send mail to arbitrary email addresses, the user could install it in a Networking Tools subfolder in the Applications folder.

Through security by designation, these measures would prevent viruses from propagating in files or email messages while minimizing the burden on users.

Cookie management

Cookies are small data records that Web sites ask browsers to retain and present to identify users when they return to the same site. They let Web sites 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.[8] It's helpful to allow users some control over when and where cookies are sent.

Many browsers address this problem by prompting users to accept or reject each received cookie. But cookies are so ubiquitous that users are constantly pestered with irritating prompt boxes when they enable this feature. To reduce the user's workload, some browsers provide additional buttons to accept or reject all cookies from the site they're currently visiting. Unfortunately, after users choose a site-wide policy, it can be hard to find and reverse the decision. Moreover, deciding to accept cookies is irrelevant because receiving cookies poses no risks; the real risks lie in sending cookies.

The missing piece of information here is whether the user, when returning to a Web site, wants to continue with the same settings and context where the last session left off. To find a better solution, we should consider how users 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 received cookies were embedded in bookmark records. Users could choose to bookmark the generic view of a

site (before logging in) or personalized view (after logging in), or even create multiple bookmarks for different sessions. The decision to activate a cookie would be incorporated into the existing task of selecting a bookmark. This isn't the same as the way people browse now, but it would be easy to explain and would not be inherently inconvenient.

Bookmarks containing cookies could be displayed with an icon to show that they are personalized. A similar icon in the browser toolbar could indicate whether the current view is personalized by a cookie. Clicking on the icon would let users select among generic and personalized views of the current site, letting users activate cookies when arriving at a site by means other than bookmarks.

This solution would eliminate the need for prompt boxes or lists of cookie-enabled or cookie-disabled sites to manage. 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: bookmarks would let them manage multiple logins or multiple suspended transactions at the same site. Security and usability would be simultaneously improved.

Phishing attacks

Forged email messages and Web sites designed to steal passwords are another serious problem. In a typical 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, styled 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 piece of information is the form submission's intended recipient. In one scam, for example, an imitation of PayPal was hosted at http://paypai.com. This problem is trickier than the preceding examples because the system has no way of knowing whether the user intended to go to http://paypal.com and was misdirected to http://paypai.com or whether the user really wanted to visit http://paypai.com. The intention resides only in the user's mind.

The designation at the heart of this security problem is the link address. But the user does not specify the address; the user merely clicks in the email message. The safest solution would therefore be to secure the email channel so users can tell whether an email comes from a trustworthy source. But until we convince most email users to use secure email, we are stuck interfacing with an untrustworthy mail system and have no user act of designation on which to hang a security decision.

In this situation, practical constraints make security by designation infeasible. We could add security by admonition at several points along the path from the email message to the impostor's form:

- The email client could make the link destination clearer by emphasizing the true domain name in the message body.
- The browser could display a user-assigned name in a reserved part of its toolbar to help identify the site hosting the form.
- As the user enters information into form fields, the browser could display the entered text in red if the user hasn't assigned a name to the receiving site before. A label below the current field could appear if the site is unfamiliar or suspicious (see Figure 3). (A similar label could warn against sending secrets over an unencrypted connection.)
- When the mouse pointer passes over a form submission button, the mouse cursor could change into a special icon annotated with the site's user-assigned name.

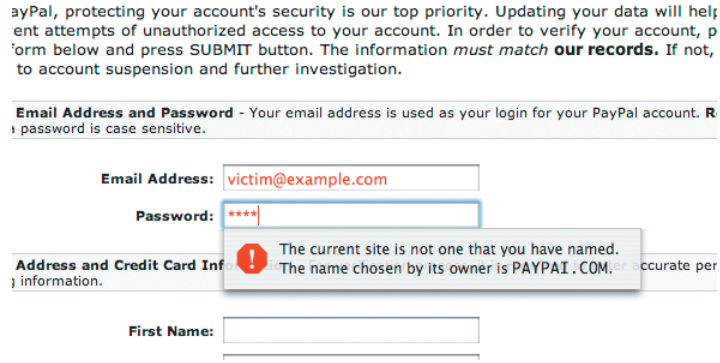

Figure 3. An admonition. As the user fills out a form, the entered text appears in red to indicate that the user has not assigned a name to this site. Without interrupting the workflow, a message informs the user of the hosting site's domain name.

Although users could overlook them, the warnings would still be more effective than prompting users on every form submission. These admonitions inform the user but don't interrupt workflow. Using forms would not require any additional effort, but phishing for passwords would be significantly less successful.

Real-world implementations

As the preceding discussion illustrates, we can achieve certain kinds of security improvements with minimal changes to the operating system, but truly strong security often requires more fundamental changes to operating systems and applications. CapDesk and Polaris are software projects that aim at different points along this trade-off.

Guidelines for secure interaction design

These 10 design guidelines are based on the actor–ability framework. Readers might find them helpful in designing and evaluating user interfaces for secure systems.

General principles

- *Path of least resistance.* The most natural way to do a task should also be the safest.
- *Appropriate boundaries.* The interface should draw distinctions among objects and actions along boundaries that matter to the user.

Maintaining the actor–ability state

- *Explicit authorization.* A user's authority should only be granted to another actor through an explicit user action understood to imply granting.
- *Visibility.* The interface should let the user easily review any active authority relationships that could affect security decisions.
- *Revocability.* The interface should let the user easily revoke authority that the user has granted, whenever revocation is possible.
- *Expected ability.* The interface should not give the user the impression of having authority that the user does not actually have.

Communicating with the user

- *Trusted path.* The user's communication channel to any entity that manipulates authority on the user's behalf must be unspoofable and free of corruption.
- *Identifiability.* The interface should ensure that identical objects or actions appear identical and that distinct objects or actions appear different.
- *Expressiveness.* The interface should provide enough expressive power to let users easily express security policies that fit their goals.
- *Clarity.* The effect of any authority-manipulating user action should be clearly apparent to the user before the action takes effect.

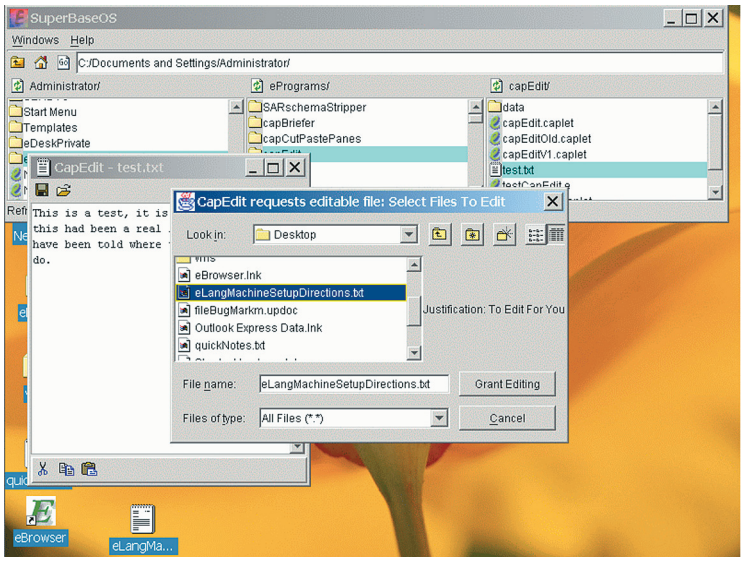

Figure 4. The user opens a file in an editor running under CapDesk.

CapDesk[9] is a capability-based desktop shell that implements security by designation, eliminating vulnerability to viruses while letting users run untrusted software in a familiar GUI environment. It explores what we can achieve by building everything from the ground up with security and usability in mind. Installing an application endows it with minimal default authorities, such as access to its own program files. The users can convey additional file access to applications by manipulating file icons and selecting files in file dialog boxes. In an earlier article, I proposed 10 guidelines for secure interaction design (see the "Guidelines for secure interaction design" sidebar for a complete list and explanation).[6] Throughout most aspects of its design, CapDesk meets all but two of these guidelines (visibility and revocability).

On the other end of the spectrum, researchers at HP Labs are developing Polaris, a safe environment for running Windows applications. Polaris explores what we can achieve by applying the concept of security by designation without changing the operating system or the applications. It provides effective protection against viruses, email worms, and spyware. In this environment, virus-scanning software's chief value is to inform the user of failed attacks. Polaris is in the early stages of prototyping, but user tests are already showing promising results. Of the 24 users who have tried Polaris, 20 found it comfortable enough to use in their daily work with the Microsoft Office suite and other applications.

Although security and usability practitioners must learn to work together to create truly secure systems, they already have more in common than might be initially obvious. Both recognize the importance of incorporating their concerns throughout the design cycle and acknowledge the need for an iterative rather than a linear design process. I have argued one step further: that security and usability not only must be considered early and iteratively, but also together.

Many obstacles prevent a perfect solution to the today's problem of security and usability, such as the installed base of operating systems and the requirement for interoperation with untrustworthy domains such as email. However, I hope the arguments and examples I present here convince readers that we can achieve sig-

nificant improvements by designing security and usability together, maintaining agreement with users' mental models, and applying security by designation wherever possible. ▢

Acknowledgments

I thank Morgan Ames, Nikita Borisov, Tyler Close, Linley Erin Hall, Marti Hearst, Mark S. Miller, Kragen Sitaker, and Marc Stiegler for their help with this article. Many of the ideas expressed here originated in discussions with them.

References

1. J. Viega and G. McGraw, *Building Secure Software*, Addison-Wesley, 2002, p. 14.
2. D. Norman, *The Invisible Computer*, MIT Press, 1998, p. 205.
3. B. Schneier, *Secrets and Lies: Digital Security in a Networked World*, Wiley, 2004, p. 12.
4. B. Lampson, "Computer Security in the Real World," *Computer*, vol. 37, no. 6, 2004, pp. 37–46.
5. S. Garfinkel and G. Spafford, *Practical UNIX and Internet Security*, 2nd ed., O'Reilly & Associates, 1996, p. 6.
6. K.-P. Yee, "User Interaction Design for Secure Systems," *Proc. 4th Int'l Conf. Information and Communications Security*, R. Deng et al., eds., LNCS 2513, Springer, 2002, pp. 278–290; http://zesty.ca/sid.
7. J.H. Saltzer and M.D. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE*, IEEE Press, vol. 63, no. 9, pp. 1278–1308.
8. B. McWilliams, "Hotmail at Risk to Cookie Thieves," *Wired News*, 26 Apr. 2002; http://wired.com/news/technology/0,1282,52115,00.html.
9. D. Wagner and D. Tribble, *A Security Analysis of the Combex DarpaBrowser Architecture*, http://combex.com/papers/darpa-review/index.html.

***Ka-Ping Yee** is a PhD student in computer science at the University of California, Berkeley. His research interests include system security, usability, and decision support systems. He is a member of the ACM, the Foresight Institute, the Python Software Foundation, the Electronic Frontier Foundation, and the Free Software Foundation. Contact him at ping@zesty.ca.*