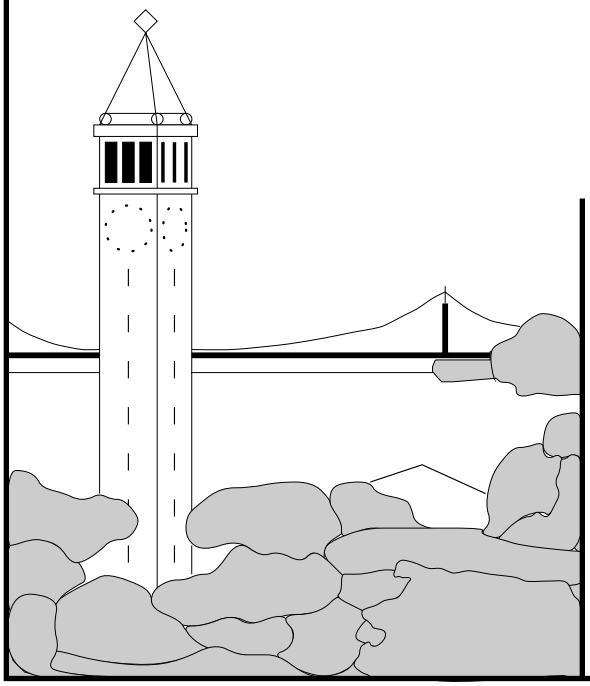
User Interaction Design for Secure Systems

Ka-Ping Yee



Report No. UCB/CSD-02-1184

May 2002

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Supported by NSF award #EIA-0122599
ITR/SI: Societal Scale Information Systems:
Technologies, Design and Applications

User Interaction Design for Secure Systems

Ka-Ping Yee
ping@zesty.ca
Computer Science Department
University of California, Berkeley

Abstract

The security of any computer system that is configured and operated by human beings critically depends on the information conveyed by the user interface, the decisions of the computer users, and the interpretation of their actions. We establish some starting points for reasoning about security from a user-centred point of view, by modelling a system in terms of *actors* and *actions* and introducing the concept of the subjective *actor-ability state*. We identify ten key principles for user interaction design in secure systems and give case studies to illustrate and justify each principle, describing real-world problems and possible solutions. We anticipate that this work will help guide the design and evaluation of secure systems.

1. Introduction

Security problems are often attributed to software errors such as buffer overruns, race conditions, or weak cryptosystems. This has focused a great deal of attention on assuring the correctness of software implementations. However, the correct *use* of software is just as important as the correctness of the software itself. For example, there is nothing inherently incorrect about a program that deletes files. But when such a program is instructed to delete files against our wishes, we perceive a security violation. In a different situation, the *inability* to command the program to delete files could also be a very serious security problem.

It follows that the security properties of any system can only be meaningfully discussed in the context of the system's expected behaviour. Garfinkel and Spafford give the definition: "A computer is secure if you can depend on it and its software to behave as you expect" [Garfinkel96]. Notice that the property of "being secure" is necessarily dependent on the meaning of "you" in this definition, which refers to the user. It is impossible to even describe security without addressing the user perspective.

Perhaps the most spectacular class of recent security problems is the e-mail virus, which is a good real-life example of a security violation in the absence of software errors. At no point in the propagation of the virus does any application or system software do anything other than exactly what its programmers would expect: the e-mail client correctly displays the message and correctly decodes the attached virus program; the system correctly executes the virus program. Rather, the problem has occurred because the expectations of the programmer became inconsistent with what the user would want.

Our purpose here is to present a way of thinking about this type of issue. Usability issues are often considered to "trade off" against security. Among many designers, there is the pervasive assumption that improving security necessarily degrades usability, and vice versa; the decision of whether to favour one or the other is typically seen as a regrettable compromise. In the end, these judgement calls are made somewhat arbitrarily because there seems to be no good answer. We believe that usability and security goals rarely need to be at odds with each other. In fact, often it is rather the opposite: a system that's more secure is more predictable, more reliable, and hence more usable. One of the significant contributions we hope to make with this paper is a coherent model for thinking about user interaction that clarifies the design process and helps one make these decisions consistently.

We have examined a variety of existing systems and have had the opportunity to discuss design challenges and user experiences at length with the designers and users of software intended to be secure. After much debate and several iterations of refinement, we have tried to distill the most productive lines of reasoning down to a succinct set of design principles that cover many of the important and common failure modes.

2. Related Work

We know of relatively few projects in computer security [Holmström99, Zurko99, Jendricke00] that have seriously emphasized user interaction issues. The Adage

project [Zurko99], a user-centred authorization service, is probably the largest such effort to date. There have been some important usability studies of security applications [Karat89, Mosteller89, Adams99, Whitten99] that demonstrate the tremendous (and often devastating) impact that usability issues can have on the effectiveness of security measures. However, to our knowledge, this paper is the first attempt to propose a structured framework for design thinking and to bring together widely applicable statements about secure user interaction as opposed to studying a single specific application or mechanism.

We acknowledge that simultaneously addressing all ten of the principles we present is a significant design challenge. But, lest they seem too idealistic to be satisfiable by a real system, it is worth mentioning that there is ongoing work on designing a secure desktop environment for personal computers that aims to meet all of these principles [Walker99], as well as an independently developed working prototype of a secure desktop environment that does a reasonable job of meeting eight of the ten principles [CapDesk].

3. Design Principles

Our criterion for admitting something as a basic principle is that it should be fairly obvious that violation of any principle leads to a security vulnerability. Our priority was to achieve good coverage, so some of the principles do overlap somewhat. They were not intended to be axioms, but rather a set of guidelines for a designer to keep in mind.

Saltzer and Schroeder's *principle of least privilege* [Saltzer75] is a basic starting point for reasoning about most of these interaction design principles. So, it may be difficult to imagine how one could meet all these principles in current operating systems that were not designed to work in a least-privilege style. It will make more sense to consider these principles in the context of a system aimed at supporting least privilege. A language-based system for enforcing security, such as Java's "sandbox", is an example of the kind of model in which one could hope to satisfy these principles. Some platforms designed around the least-privilege concept include the E scripting language [E], KeyKOS [Hardy85], and EROS [Shapiro99].

First, we'll present just the design principles themselves, and then build up a structured set of concepts that allows us to explain the thinking behind each of them in more detail. In the statement of these principles, we have used the term "actor" to mean approximately "user or program", but we will need to define this term more precisely below. When we say "authority", we just mean the ability to take a particular action.

Path of Least Resistance. To the greatest extent possible, the natural way to do any task should also be the secure way.

Appropriate Boundaries. The interface should expose, and the system should enforce, distinctions between objects and between actions along boundaries that matter to the user.

Explicit Authority. A user's authorities must only be provided to other actors as a result of an explicit action that is understood by the user to imply granting.

Visibility. The interface should allow the user to easily review any active authority relationships that would affect security-relevant decisions.

Revocability. The interface should allow the user to easily revoke authorities that the user has granted wherever revocation is possible.

Expected Ability. The interface must not generate the impression that it is possible to do something that cannot actually be done.

Trusted Path. The interface must provide an unspoofable and faithful communication channel between the user and any entity trusted to manipulate authorities on the user's behalf.

Identifiability. The interface should enforce that distinct objects and distinct actions have unspoofably identifiable and distinguishable representations.

Expressiveness. The interface should provide enough expressive power (a) to describe a safe security policy without undue difficulty; and (b) to allow users to express security policies in terms that fit their goals.

Clarity. The effect of any security-relevant action must be clearly apparent to the user before the action is taken.

3.1. The User and the User Agent

Thus far, we have mentioned "the user" several times, so it is necessary to precisely define what we mean by the user. For the purpose of the rest of this discussion, the user is a person at a computer using some interface devices such as a keyboard, mouse, and display.[1]

We are concerned with the software system that is intended to serve and protect the interests of the user, which we will call the "user agent". On a single-user

[1] It is sometimes important to be able to consider an organization, rather than an individual, "the user" of a piece of software. It is interesting to re-examine the above design principles from the perspective of an organization, although some of the issues raised might not be addressed by what one would typically call a user interface. It is also interesting to look at the design principles from the perspective of a software program. Although some principles might not apply (because programs do not have fallible memories and perception systems like people do), many of them still make sense (the ability to make decisions still depends on the availability of sufficient information, the principle of least privilege still holds, and so on).

system, the user agent is the operating system shell (which might be a command line or a graphical shell), through which the user interacts with the arena of entities on the computer such as files, programs, and so on. On a multi-user system, that arena expands to include other users, using their own user agents to interact within the same arena.

When the system is networked, say to the rest of the Internet, there is a new, second level of interaction. Now, the arena of the single computer is nested within the larger arena of the Internet. A new kind of user agent (such as an e-mail client or a Web browser) now represents the user's interests in that larger arena of interacting entities (which again includes other users with their own user agents). But in the smaller arena of the single computer, a Web browser is merely one of the participants, and the user's interactions with it are mediated by the lower-level user agent, the system shell. The Web browser might be used to contact yet a third user agent, such as a Web-based interface to a bank, operating in yet a third arena (of financial transactions among account holders), and so on.

We point out this distinction here mainly to help avoid confusion. We will not directly address the issue of communicating through multiple levels of user agents here; it's simpler to select an arena and think within the context of just one level at a time. The rest of this paper discusses the design of any user agent serving a user. The principles of design we present can apply to all kinds of users, including not just end users of application software, but also system administrators and programmers, using whatever software they use for their tasks. Different users will have different expectations and needs, so the design of any secure system must begin with a clear understanding of those needs. This is why the principles are stated in terms of what the user perceives and what the user expects.

3.1.1. Principle of the Path of L

In the real world, there is often no relationship between how safe or unsafe actions are, and how easy or hard they are. (It takes much more concentration to use a hammer safely than unsafely, for example.) We have to learn, by being told, by observing others, and often by making many painful mistakes, what ways of doing things are safe. Sometimes, through the design of our tools, we can make it a little easier to do things safely. Most food processors have a switch that allows them to operate only when the lid is closed. On power drills, the key for opening the drill chuck is often taped to the power cord so that unplugging the drill becomes a natural prerequisite to changing the drill bit. In both of these cases, a bit of cleverness has turned a safety precaution into a natural part of the way work is done, rather than an extra easily forgotten step.

Most users do not spend all their time thinking about security; rather, they are primarily concerned with accomplishing some useful task. It is human nature to be economical with the use of physical and mental effort, and to tend to choose the "path of least resistance". This can sometimes cause the user to work against security measures, either unintentionally or intentionally. If the user is working against security, then to a large extent the game is already lost. Hence, the first consideration is to keep the user's motivations and the security goals aligned with each other.

There are a number of aspects to this. First, observe that the ultimate path of least resistance is for the user to do nothing; that is, to leave the system in its default state. Therefore, the default settings for any software should be secure (this is Saltzer and Schroeder's principle of "fail-safe defaults" [Saltzer75]). It is unreasonable to expect users to read documentation in order to learn that they need to change many settings before they can run software safely.

Second, consider how a user might work against security measures unintentionally. The user interface leads users to do things in a certain way by virtue of its design, sometimes through visual suggestion, and sometimes in other ways. The word "affordance" was introduced by J. J. Gibson [Gibson77] to refer to the properties of things that determine how they can be interacted with. Don Norman applied this concept to interaction design [Norman88]. In computer user interfaces, the behaviour of users is largely guided by what affordances they perceive. They decide what actions are available to them by observing whether particular things appear as though they ought to be clicked or dragged or typed at, and so on. For example, suppose an icon of a lock can be clicked to request detailed security information. If the icon is not made to look like something that's supposed to be clicked, the user might never notice that this was an available action, eliminating the usefulness of the feature.

Third, consider whether a user might subvert security intentionally. If operating securely requires too much effort, users might decide to circumvent or ignore security measures even while completely aware that they are doing so. Therefore, there is a security risk in a system where the secure patterns of usage are inconvenient: each added inconvenience increases the probability that the user will decide to operate the software unsafely.

All of these aspects can be summarized by the **principle of the path of least resistance**: the natural way should be the secure way.

Sometimes the desire to make things easy and natural may seem to be in conflict with the desire to make things secure. However, these goals are truly in conflict less often than one might think, as shown by the file browser example we will explain in Section 3.5.1.

Making security tighter usually has to do with getting more specific information about what goal the user wants to accomplish so we can do it more carefully. Often this information is already available in the user's actions; it just needs to be applied consistently to improve security. There remain a few situations where, for the sake of security, it may be absolutely necessary to introduce a new inconvenience. When this is the case, provide a payoff to offset the cost of the new inconvenience, by making productive use of the extra information the user is asked to provide.

For example, consider a multi-user system that requires people to log in with a username and a password before they can do anything. Logging in is an extra step that is necessary for security, but has little to do with the user's intended task. However, we can use that extra information to personalize the user's experience—providing them with their own custom desktop, menu of favourite programs, and so on—to justify the added inconvenience. This helps to keep the user from trying to circumvent the login process (or choosing to use a software system that doesn't have one).

3.2. Objects, Actors, and Actions

Before explaining the next few principles, we need to introduce a few more concepts.

In order to productively interact with the world around us, we build a mental model of how it works. This model enables us to make predictions about the consequences of our actions, so that we can make useful decisions. In the model, most concepts fall within the two fundamental categories of *objects* and *actions*. This division is reflected in the way that practically all languages, natural or invented, draw the distinction between nouns and verbs.

Some objects are relatively inert. The way they interact with other things is simple enough to be modelled with physical laws. For instance, if I shove a mug off the edge of a table, I expect it to fall to the ground. In Dennett's terminology, my model adopts the "physical stance" [Dennett87] toward the mug. It is usually straightforward to work with such objects because we can predict quite precisely what they will do. On a computer, one might consider a text file an example of such an object. We can do things to the text file (say, copy it or delete it) that have simple, predictable consequences, but the file takes no actions of its own.

Some objects have their own behaviours; we will call such objects *actors*, since they are capable of taking actions of their own. Even though such objects exist in the physical world and still follow physical laws in principle, their behaviour is too complex to model using only physics. It is impossibly difficult to predict exactly what an actor will do; instead, we can only proceed by estimating reasonable bounds on what the actor will do.

To a computer user, an application program is an actor. There are some expectations about what the program will do, and some established limits on what it should be able to do, but no user could know in detail exactly what program instruction is being executed at a given moment. Even though the operation of the program may be completely deterministic, we cannot take a physical stance toward it because it is too complex. Instead, we must model the program based on our understanding of the purpose for which it was designed (this is taking the "design stance" [Dennett87]).

Other users are also actors. However, rather than having been designed for a purpose, their behaviour is directed by their own motivations and goals. As they are conscious entities, we model their behaviour in terms of our knowledge of their beliefs and intentions; that is, we adopt the "intentional stance" [Dennett87] toward such entities.

Incomplete knowledge of the design, beliefs, or intentions of an actor produces uncertainty. We limit this uncertainty by applying the physical stance: while I am inside a locked house, for example, I have no need to model the intentions of any people outside the house because I am relying on the physical properties of the house to keep them out of my model.

Building models of actors is something we humans are very good at, since we have been learning how to do it all our lives. Bruce and Newman have examined in detail how the comprehension of "Hansel and Gretel" requires us to model actors, actors' models, actors' models of other actors' models, and so on, many levels deep [Bruce88]—yet such complex modelling is a routine skill for young children. There is also significant evidence from computer-human interaction research that people perceive computers as "social actors" [Nass94] even though machines do not actually possess human motivations. Both of these reasons suggest that we indeed form our mental models of computers in terms of actors and actions.

Given this foundation, we can now formulate a more precise interpretation of Garfinkel and Spafford's definition of computer security. Our new definition is: "A system is secure from a given user's perspective if the set of actions that each actor can do are bounded by what the user believes it can do."

3.3. The System Image and the User Model

When a designer creates a system, the designer does so with some model in mind. But the designer doesn't get to communicate directly with the user. Rather, the designer decides how the system will work, the system presents an image to the user, and the user builds a model from interacting with the system. Communication of the model occurs only via this *system image*.

Figure 1 illustrates this flow of information. In the next few sections, we'll look at some important features of the system image and the user model.

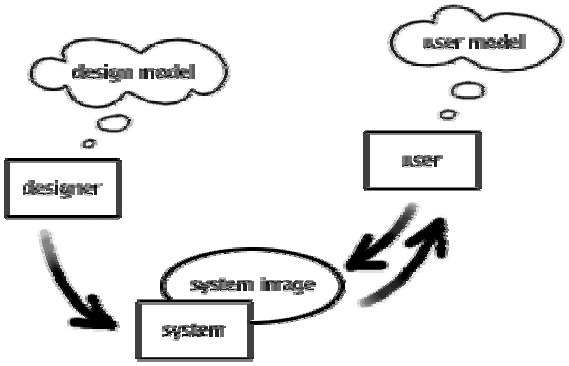


Figure 1. Designer, system, and user (from [Norman88]).

3.4. Aggregation

The actual working of a computer system is extremely complex and involves a tremendous number of small components and operations. There may be many thousands of objects involved and an unlimited variety of possible actions. To make the system comprehensible, the system image aggregates these objects and actions into a smaller number of units.

Objects may be grouped by related concept or purpose. All the individual bytes of a file are usually taken together, given a single name, and presented as a single manipulable object. Actions may be grouped by concept, by locality in time, or by causality relationships. For example, while a request to open a Web page may involve many steps (looking up a hostname, opening a network connection, sending a request, downloading the response, parsing the response, and then proceeding to do the same for any embedded images), it is presented as a single action. (The modelling notation in [Bruce88] includes an abbreviation called "ByDoing" for this kind of aggregation.)

Most user interfaces allow the user to control some grouping in order to reduce their own mental effort. For instance, in most desktop operating systems, one can move a collection of files into a directory, and then move, copy, or delete the entire directory with a single operation. The grouping is up to the user: that is, one can perform *subjective aggregation* [Miller00] on the file objects. Systems that support end-user programming features, such as macros, allow the subjective aggregation of several actions into a single action.

3.4.1. Principle of Appropriate Boundaries

Aggregation is important because it defines the terms in which authorities can be expressed. The user's model deals with concepts such as "actor X is capable of performing action Y to object Z". The boundaries of the objects and actions are defined by observing the system image. The system image conveys these boundaries through the ways that the user can identify objects, communicate with actors, take actions, and so on.

Here is an example to demonstrate the significance of choosing these boundaries. Consider the basic intuition that a secure operating system should allow the user to control the granting of authorities to applications. If an application program spawns multiple processes, does this then mean that the user must separately grant authorities to each process? Or if a program relies on software modules or shared libraries to take care of work on its behalf, should the user have to separately control the authorities of every module? No: we resolve this apparent usability crisis by declaring that the boundaries in the system image (which are also the boundaries of authority control) should be consistent with distinctions the user actually cares about. Any boundary that could have meaningful security implications to the user should be visible, and those that do not should not be visible.

In short, this is the **principle of appropriate boundaries**: the interface should distinguish objects and actions along boundaries that matter to the user. If the distinctions are too detailed, there is an increased risk that users will overlap or leave out specifications. On the other hand, if the boundaries are too few, users will be forced to give away more authority than they intend. The right distinctions can be discovered by asking oneself if there are situations where the user would ever want to manipulate one authority independently of another, to grant an authority to one actor but not another, to permit access to one resource but not another, and so on.

Supporting good distinctions sometimes places requirements on the software system behind the user interface. In the case of our example, since it would be infeasible to insist on separate control of authorities for each software component, the system should support the safe aggregation of software components into useful conceptual units (that is, applications), such that reasoning about the authorities of an application as a unit still holds valid. The system image should present boundaries between different applications; consequently,

whenever two applications use the same software module, that module should be unable to convey authority between the applications.

3.5. The Actor-Ability State

Among other things, the user model contains knowledge about all the actors' abilities. More specifically, at any point in time, the user knows of a finite set of actors $A = \{ A_0, A_1, A_2, \ldots A_n \}$ that can have an effect on the system, where A_0 is the user and there are *n* other actors. Each actor A_i is associated with an alleged set of potential actions, P_i. One can think of P_i as the user's answer to the question, "What can this actor do that would affect something I care about?" The knowledge of actors and abilities then consists of $< A = \{ A_0, A_1, A_2, \ldots A_n \}, P = \{ P_0, P_1, P_2, \ldots P_n \} >$. We will call this subjective information the user's *actor-ability state*.

Since the user believes that P_0 is the set of available actions he or she can perform, the user will always choose to do actions from that set. In order for the user to choose things that are actually possible to do, P_0 should be a *subset* of the user's actual abilities.

Since the user believes that P_i (for $i > 0$) is the set of available actions some other actor A_i can perform, the user expects that any action taken by A_i will be a member of P_i. Therefore, to uphold this expectation, P_i must be a *superset* of that actor's actual abilities.

3.5.1. Principle of Explicit Authority

It is essential to keep the actor-ability state in the user's model accurate at all times, since the user will make security-relevant decisions based on this state. To stay synchronized with reality, the user must be in control of any changes that would affect the actor-ability state. This is the **principle of explicit authority**: any explicit action must be taken to grant new authorities to other actors. More precisely, since the user's actor-ability state is a set of *bounds* on each actor's abilities (rather than a precise enumeration of each specific ability), we require an explicit action to be taken in order for the set of available actions for any actor to come to exceed the bounds in the user's current actor-ab state. Or, in other words, we must maintain the constraint that P_i is a superset of the actual abilities of A_i, for all other actors (that is, for $i > 0$).

Explicit authority is perhaps the most basic requirement for controlling authority in any system. In current systems, applications often have authorities to resources such as the network and filesystem without ever having been explicitly granted these authorities. Explicit authority is a direct descendant of Saltzer's principle of least privilege. Requiring each authority to be explicitly granted increases the likelihood that actors will operate with the least authority necessary. Without such a restriction, the user becomes responsible for finding a potentially unlimited set of implicitly granted authorities to disable before the system is safe to use.

At first glance, it may seem that the principle of explicit authority is in conflict with the principle of the path of least resistance. Does the principle of explicit authority mean that we must now constantly intercept the user with annoying security prompts to confirm every action? In fact, most of the time, we do not need to ask for extra confirmation; the user already provides plenty of information in the course of performing the task. We necessary need to make sure that the system honours the manipulations of authority that are already being communicated. For example, if the user asks an application to open a file and selects a file from a file browser, it is already clear that they expect the application to read the file. No further confirmation is necessary. The single act of selecting the file should convey both the identity of the chosen file and the authority to read it [Hardy88].

We can judge when explicit authority is n
the basis of the user's expectations. For example, if there is a window on the screen that clearly belongs to an application, users will expect the application to draw in the window. However, it would certainly be unexpected for the application to spontaneously delete the user's personal documents. Just as it requires an explicit action by the user to instruct the computer to delete personal files, so should it require explicit action by the user for any program to acquire the ability to delete them.

The judgement of what authorities should be explicit should be based on the potential consequences, not on the technical difficulty of the decision to be made. *Any* authority that could result in unexpected behaviour should be controlled by the user. If the user cannot readily understand the consequences of granting an authority, then that authority should never be granted at all, not merely hidden under some "Advanced" section of the interface. If a truly necessary authority seems to require an unusual degree of technical knowledge, then the model presented to the user probably needs to be rethought in terms that can be understood.

3.5.2. Principle of Visibility

If the actor-ability state begins as a known quantity (for example, we have a safe minimal set of authorities, such as allowing applications to draw within labelled, bounded areas of the screen), *and* we are in control of each change in state, then in theory we have enough information to ensure that our state is always accurate.

However, there will often be situations where one has to come upon a new system in an unknown state. Moreover, it is unreasonable to expect a user to keep a perfect record of all grantings of authorities; human memory is fallible and limited in capacity. Therefore, we must enable users to update the actor-ability state in their heads at any time. That is, the system must support the **principle of visibility**. This is not to say that the interface should display all the low-level authorities of all the components in the system as a debugger would. Rather, it should show the right information for the user to ascertain the limits of what each actor can do, and should do so in terms of actors and actions that fit the user's model.

Visibility of system state is advocated by Jakob Nielsen as essential for usability in general [Nielsen94]. Likewise, visibility of authorities is n
understand the security implications of their actions. Since such authorities come about as a result of the user's granting actions, it makes sense to show the actor-ability state in terms of those granting actions. Past granting actions that have no effect on the current state (access given to an applic
terminated) need not be visible. It is helpful to be able to identify authorities by inspection of either the holder or the resource to which the authority gives access. Without visibility of authorities, any application that gains an authority could retain and use that authority undetected and indefinitely, once the user has forgotten about the granting action.

3.5.3. Principle of Revocability

To keep the actor-ability state manageable, the user must be able to prevent it from growing without limit. Therefore, wherever possible, the user should be allowed to revoke granted authorities; this is the **principle of revocability**.

Another, stronger argument for facilitating revocation is the need to accommodate user error. It is inevitable that people will make mistakes; any well-designed system should help recover from them. In the context of granting authorities, recovery from error amounts to revocation. One might intentionally grant an authority to an application and later discover that the application is misguided or malicious; or one might inadvertently grant the wrong authority and want to correct the mistake. In both of these cases, the granting decision should be reversible. Note that revocation prevents *further* abuse of an authority, but it is rarely possible to undo any damage caused by the abuse of an authority while it was available. Therefore, interfaces should be careful not to draw an analogy between "revoke" and "undo"; instead, "revoke" is better described as "desist".

3.5.4. Principle of Expected Ability

When we introduced the actor-ability state, we mentioned that P_0 should be a subset of the user's actual abilities. This can also have security consequences. In the course of performing tasks, users sometimes make decisions based on the expectation of future abilities. If these expectations are wrong, the user might make the wrong decision, with serious security consequences. In some situations, the false expectation of an ability might give the user a false sense of security, or cause the user to make a commitment that cannot be fulfilled. Hence the **principle of expected ability**: the interface must not give the user the false impression of an ability.

For example, suppose the user is working in a system where granted authorities are usually revocable. If the user comes across an authority for which revocation is not supported, the interface should make it clear that the authority cannot be revoked, as this could affect the user's decision to grant it.

Or suppose the interface makes the claim that it is possible to prevent an application from broadcasting information on the Internet. The user could then decide to reveal sensitive information, based on this expectation. If the system in fact cannot enforce the restriction, the user has been misled into enabling a security violation.

3.6. Input and Output

Observation and control is conveyed through input and output, so the ability to use a system securely relies on the integrity of the input and output channels.

3.6.1. Principle of the Trusted Path

The most important input and output channels are those used to manipulate authorities; if these channels can be spoofed or corrupted, the system has a security vulnerability. Hence the **principle of the trusted path**: the user must have an unspoofable and incorruptible channel to any entity trusted to manipulate authorities on the user's behalf.

The authority-manipulating entity could be a number of different things, depending on the domain. In an operating system, the authority-manipulating entities would be the operating system and user interface components for handling authorities. Microsoft Windows, for example, provides a trusted path to its login window by requiring the user to press Ctrl-Alt-Del. This key sequence causes a non-maskable interrupt that can only be intercepted by the operating system, thus guaranteeing that the login window cannot be spoofed by any application. In a language system for running untrusted code, such as Java, this issue also needs to be addressed.

3.6.2. Principle of Identifiability

The ability to *identify* objects and actions is the first step in proper communication of intent. When identity is threatened, either by inadvertent collision or by intentional masquerading, the user is vulnerable to error. Identification has two aspects: *continuity* (things which are the same should appear the same) and *discriminability* (things which are different should appear different).

That something is perceived to have an identity depends on it having some consistency over time. When we see an object that looks the same as something we saw recently, we are inclined to believe it is the same object. If an untrusted program can cause an object to look the same as something else, or it can change the appearance of an object in an unexpected way, it can produce confusion that has security consequences. The same is true for actions, in whatever way they are represented; actions are just as important to identify and distinguish as objects.

Note that it is not enough for the representations of distinct objects and actions to merely *be* different; they must be *perceived* by the user to be different. For example, a choice of typeface can have security consequences. It is not enough for two distinct identifiers to be distinct strings; they must be displayed with visually distinct representations. In some fonts, the lowercase "L" and digit "1" are very difficult to distinguish. With the use of Unicode, the issue is further complicated by characters that combine to form a single accented letter, as this means many different character sequences can be rendered identically on the screen.

Tessiveness Effective communication of intent is vital, and we cannot assume that objects will give themselves unique and consistent representations, identifiability is something that must be ensured by the system. This gives us the **principle of identifiability**: we must enforce that distinct objects and distinct actions have unspoofably identifiable and distinguishable representations.

3.6.3. Principle of Exp

Sometimes a security policy may be specified explicitly, as in a panel of configuration settings; other times it is implied by the expected consequences of actions in the normal course of performing a task. In both cases, there is a language (consisting of settings or sequences of actions) through which the user expresses a security policy to the system.

If the language used to express security preferences does not match the user's model of the system, then it is hard to set policy in a way that corresponds with intentions. In order for the security policy enforced by the system to be useful, we must be able to express a safe policy, and we must be able to express the policy we want. This is the **principle of expressiveness**.

3.6.4. Principle of Clarity

When the user is given control to manipulate authorities, we must ensure that the results reflect the user's intent. We rely on software correctness to enforce [illegible] the authorities available to an actor; but the correctness of the implementation is irrelevant if the policy being enforced is not the one the user intended. This can be the case if the interface presents misleading, ambiguous, or incomplete information.

The interface must be clear not only with regard to granting or revoking authorities; the consequences of any security-relevant decision, such as the decision to reveal sensitive information, should be clear. All the information n

accurate and available before an action is taken, not afterwards, when it may be too late; this is the **principle of clarity**.

An interface can be misleading or ambiguous in non-verbal ways. Many graphical interfaces use common widgets and metaphors, conditioning users to expect certain unspoken conventions. For example, round radio buttons usually reflect an exclusive selection of one option from several options, while square checkboxes represent an isolated yes-or-no decision. The presence of an ellipsis at the end of a menu command implies that further options need to be determined before an action takes place, whereas the absence of such an ellipsis implies that an action will occur immediately when the command is selected.

Visual interfaces often rely heavily on association between graphical elements, such as the placement of a label next to a checkbox, or the grouping of items in a list. Within a dialog box of security settings, for instance, we might be relying on the user to correctly associate the text describing an authority with the button that controls it. The Gestalt principles of perceptual grouping [Wertheimer23] can be applied to evaluate and improve clarity:

- Proximity: items near each other are seen as belonging together.
- Closure: line breaks and form discontinuities are filled in.
- Symmetry: symmetrically positioned and shaped objects are seen as belonging together.
- Figure-ground segregation: small objects are seen as the foreground.
- Continuation: objects that follow a line or curve tend to be seen as belonging together.
- Similarity: similar shapes belong together.

3.7. Summary

In order to have a chance of using a system safely in a world of unreliable and sometimes adversarial software, I need to have confidence in the following statements:

- Things don't become unsafe "all by themselves". (Explicit Authority)
- I can know whether things are safe. (Visibility)
- I can make things safer. (Revocability)
- I don't choose to make things unsafe. (Path of Least Resistance)
- I know what the system can do for me. (Expected Ability)
- The system enables me safely to do what I want. (Appropriate Boundaries)
- I can tell the system what I want. (Expressiveness)
- I know what I'm telling the system to do. (Clarity)
- The system protects me from being fooled. (Identifiability, Trusted Path)

4. Case Studies

In the following sections, we analyze some security problems that arise from usability issues in real-life applications, and show how the a
apply. We present these case studies to justify our claim that each of these design principles is in fact a significant consideration and to show that each one can be applied in practice to guide design for better security.

4.1. ActiveX and Code Signing

Problem: When an untrusted ActiveX control is downloaded from a Web page, its digital certificate is presented to the user for approval. Most of the time, users accept certificates without paying them much attention, even from unknown sources. Once a malicious ActiveX control would have full a machine, and could easily wipe out or overwrite anything on the hard drive. Although one could try to legally pursue the source identified on the certificate, the damage is already done. If the damage is done quietly (say, an alteration to an important accounting file), it might not be discovered until much later.

Further, consider a more subtle and insidious attack in which an ActiveX control appears to perform its intended function, but meanwhile silently modifies the certificate-checking behaviour of the operating system. It could make the certificate checker function properly for the next 99 times and then destroy the hard drive on the 100th ActiveX control downloaded; or it could even have it destroy the hard drive when it sees a certificate signed by a particular other party that the attacker wants to incriminate. This kind of delayed attack would be virtually impossible to trace.

Analysis: Although the cryptography behind code signing is perfectly sound, its effectiveness is diminished because few users ever check the validity of certificates in practice. Users find that it takes too much effort to even read the certificate, and most don't know how to verify the fingerprint to ensure that it matches the claimed certifying authority. In the ActiveX scheme, the easiest action — to simply click "Okay" and proceed — is also the most dangerous. It is clear that this scheme was designed without regard to the **path of least resistance**.

Solution: The security of the system should not rely on an assumption that users will always expend time and effort on security checks. By default, downloaded code should run with limited authority. Granting extra authorities to a downloaded program should require special action from the user, and in no event should the program be allowed to modify the operating system.

4.2. Java Applet Permissions in IE

Problem: The dialog for editing Java permissions in Microsoft Internet Explorer 5, shown in Figure 2, presents two independent options regarding files: "Access all files" and "D
gives universal permission to read and write any file; "Dialogs" permits applets to show a file chooser dialog.

There is no way to allow a
even to allow reading without writing. To accomplish almost any useful creative task, one often needs to save work locally. But permitting a
applet to destroy the entire file system, which is clearly almost never desirable. There is also no way to distinguish individual applets; Internet Explorer groups Web sites into four zones, and all applets from all sites in a zone are treated with the same policy.

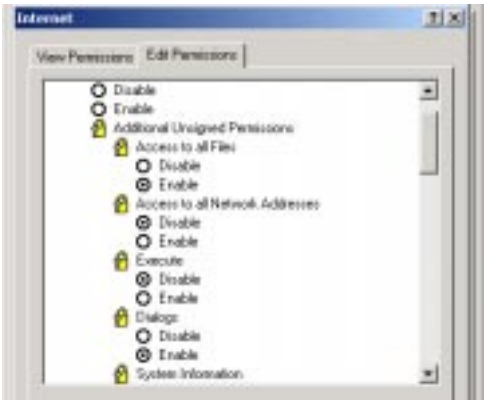


Figure 2. Java security settings in Internet Explorer.

Analysis: The system image has failed to provide **appropriate boundaries** between distinct file objects, between the distinct actions of reading and writing, or between distinct actors (that is, different Java applets). The permissions dialog also violates the principle of **clarity** by presenting confusing and vague controls. None of the permissions are explained in any detail: it is not made clear that "Access to all files" includes modification or destruction of files, nor is it explained just what kinds of dialogs the "Dialogs" setting affects.

Solution: The problem of clarity could be addressed in isolation by adding better descriptive text to the dialog box. However, a much better solution would be to eliminate the permissions dialog altogether and grant file access at the
needs to read or write one file, then the Java language system should present a file chooser that clearly indicates whether read or write access is requested, and the user's selection of a file should convey the authority to read or write it. This change would greatly improve both security and usability at the same time.

4.3. E-mail and Macro Viruses

Problem: The "Melissa" virus was first reported on 26 March 1999 and within three days it had infected more than 100,000 computer systems [CA-1999-04]. Despite widespread publicity about Melissa and increased demand for computer security measures, most computers remained unprotected. Over a year later, in May 2000, a similar virus known as "Love Letter" spread even more rapidly; it was estimated to have infected millions of computer systems within just a couple of days [CA-2000-04]. The Love Letter virus did much more damage than Melissa, destroying most of the image and music files on each machine.

Analysis: The permissive nature of Microsoft Windows made it trivially easy for these viruses to infect other computers and destroy files. Here are some of the authorities abused by these viruses, none of which are necessary to the reading of a typical e-mail message:

1. Upon a request from the user to examine an attachment, the enclosed script or macro was given permission to execute.
2. The script or macro was allowed to discover all the files on the machine and overwrite them.
3. The script or macro was allowed to read the Microsoft MAPI address
4. The script or macro was allowed to command Microsoft Outlook to send out mail.

Item 1 is a violation of the principle of **clarity**. The recipient did take explicit action to see the contents of the attachment, but was misled about the potential consequences. In the user's mind, the desired action is to *view* the attachment; instead, the action actually taken is to *execute* it. In the case of Melissa, the attachment was just a Microsoft Word document, and few users were aware that a document could actively damage the system upon being opened. In the case of the Love Letter worm, the operating system hid the ".vbs" extension on the filename "LOVE-LETTER-FOR-YOU.TXT.vbs" so that the attached file appeared to be a text file; again, the recipient had no obvious warning that opening the file could damage the system.

Items 2 through 4 are violations of the principle of **explicit authority**. A typical e-mail message never needs to be given the ability to trigger the transmission of further mail, yet the e-mail client extended these permissions freely to the attachment without any explicit action from the user. Neither the e-mail client nor Microsoft Word need permission to overwrite arbitrary files on the disk at all, and the operating system should not have granted them this permission without explicit action from the user.

Solution: When an action will cause the creation of a new actor, as in item 1, the interface should make it clear that this will happen. The system should follow the principle of explicit authority, and avoid giving out the authority to destroy files or send e-mail unless the user specifically authorizes them.

4.4. Back Orifice

Problem: Many operating systems, including Windows and Unix, make it easy for programs to run "in the background" in a way that is invisible to the typical user. Although there is a way to view a list of processes, the user must first suspect the existence of a harmful process before he or she can discover and disable it.

One of the most widely publicized examples of a potentially harmful background process is the "Back Orifice" program released by a group named Cult of the Dead Cow in mid-1998. Back Orifice can be used to remotely control a computer running Windows, including capturing keystrokes and images of the screen and transmitting or modifying its files.

Analysis: Microsoft responded to concerned customers with a security bulletin claiming that Back Orifice "does not expose or exploit any security issue regarding Windows, Windows NT, or the Microsoft BackOffice suite of products" [Microsoft98]. The bulletin further implied that the user is fully responsible for evaluating the security consequences of any software they download or install. This reflects a narrow, developer-centric point of view. The expectation that one will never make any mistakes while installing or configuring software is clearly unreasonable, even for the most experienced system administrators.

Even if we assume that the user understands exactly what Back Orifice does, there certainly is a security issue regarding Windows demonstrated here. Suppose the user launches Back Orifice accidentally (perhaps two nearby clicks get interpreted as a double click); or the user launches it intentionally but forgets that it is running; or the user closes all visible Back Orifice windows, believing that the program has terminated when in fact it has not. The issue is one of **visibility**: Windows freely allows programs like Back Orifice to run in the background, monitoring and controlling the user's machine, without ever making the user aware of the program's existence. The Windows task bar, where users expect to see the set of currently running programs, shows no evidence of Back Orifice. Hence, an entire actor can be missing from the user's actor-ability state.

Solution: If the system is changed to ensure that all actors were always visible, a user would be aware of Back Orifice whenever it is running, and could avoid doing sensitive work or take steps to turn it off.

4.5. Software Installation and Maintenance

Problem: Today it is common practice on most end-user systems to treat the installation of software programs and device drivers as a kind of electronic Russian Roulette. After a user downloads software components or purchases hardware with packaged software drivers, the installation process is a complete mystery. There is little or no indication what resources are being given to the new software, what global settings are being modified, or how to restore the system to a stable state if the installation fails.

The designers of software are typically most concerned with making their own software work well, and are less worried about damaging the functionality of previously installed (and possibly competing) software. As a result, the installation of new software often destroys configuration parameters that the user may have taken great pains to set up, and can leave the system in a state where some hardware devices can no longer be used with any other applications. Frequently the only available recourse is to try to guess what settings might be changed and write them down on paper in advance.

Analysis: Software packages should not be able to take over hardware control without the user's permission. This is a violation of the principle of **explicit authority**. Just as important is the user's ability to audit and revoke authorities so that he or she can confidently restore the system to a working state. It might not be possible to refuse authority initially (one can't know whether software works before having tried it), so the granting of authorities must be reversible. This is the principle of **revocability**.

Ultimately the user must retain control over the system's resources, and decide which software gets to control what. Multiple software products that want to use the same resources should be able to coexist in such a system, where the user can easily switch from one to another. To do so, the user must be able to know where that control lies, and to revoke and reassign it at will.

Solution: Consider the analogy that installing a new component of a stereo system is not that different from installing software. Suppose I purchase a new speaker. I install it by connecting it to my radio, thus allowing the radio to employ the speaker to generate sound. At any time, I can revoke all authority to control the speaker by picking up the speaker and disconnecting any cables leading to it; then I have complete confidence that the speaker is unaffected by the stereo system, and I am free to take it and use it elsewhere. Ideally, one would like installing, interchanging, and removing software components to approach this level of simplicity.

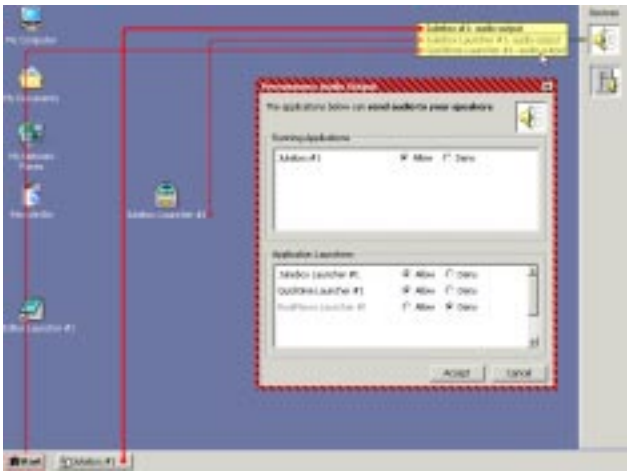


Figure 3. A possible interface for revoking authorities.

Here is one possible interface design that might begin to address these issues, based on the stereo-system analogy. Figure 3 shows a mocked-up screenshot of a system where a music-playing program named "Jukebox" has been installed. There is a Jukebox *launcher* on the desktop, and the user has also started one running instance of the Jukebox. In the figure, the user has inspected the speaker device on the right, and the display shows that one running program and two launchers have access to the speaker. Arrows connect the program (on the taskbar) and the launchers (one on the desktop and one buried in the Start menu) to the speaker icon. A dialog box lets the user revoke any of these authorities.

The general problem of software installation is large and complex. Although a complete solution is outside the scope of this paper, this design example should help to demonstrate that it is possible to make some progress.

4.6. Namespace Collisions in the Unix Shell

Problem: Unix users are often warned against putting "`.`", which refers to the current directory, in their `PATH` environment variable. Suppose that a malicious user creates a shell script called "`ls`" that tries to delete every file on the disk. If a user with "`.`" in their `PATH` happens to change into the directory containing this script and tries to list its files by typing "`ls`", they will end up destroying all their files.

Analysis: In this example, the interface has allowed **identifiability** to be violated. When objects are referenced by name, non-unique names can cause confusion. A global namespace increases the likelihood that multiple objects will have the same name.

In response to a command such as "`ls`", most Unix shells search all of the directories listed in the `PATH` variable for a file named "`ls`". For instance, the directory `/bin` will be in the `PATH` as it contains many basic programs, including "`ls`". This searching behaviour produces the confusing effect of multiple overlapping global namespaces, and introduces a potential security weakness. Programs specified at the command line are not uniquely identified; if there are two programs with the same name on the system and one is malicious, it is possible that the malicious program will be invoked where the safe one was intended.

If the `PATH` contains only restricted-access directories, these directories are known to contain only safe programs, and the programs are all known to have unique names, then there is no problem. But the system does not enforce any of these restrictions.

Solution: Although this example is specific to Unix, it provides motivation for avoiding such discriminability problems in general. Our example shows that one should avoid using names in a global namespace to refer to objects. Selecting an object directly, rather than by giving its name, is also better from a usability standpoint because recognition is easier than recall [Nielsen94].

When names are absolutely necessary, unique names should be enforced. One solution to the `PATH` problem is to use only *local* namespaces instead of global ones. If, instead of a `PATH` variable, each user had their own directory containing hard links to the programs they use, the confusion described here could not occur.

4.7. Website Password Prompts

Problem: Suppose that Alice and Bob both run Web sites requiring user authentication. Both sites are hosted on the same server, `some-Web-host.org` (perhaps Alice and Bob use the same free hosting service, for instance). If we open two browser windows, one at Alice's site and one at Bob's site, and then proceed to enter our passwords at both sites, two password prompts will appear, as in Figure 4.

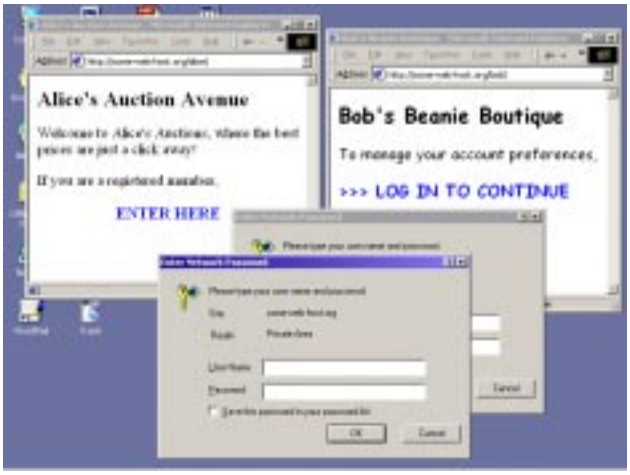


Figure 4. Two browser windows ask for passwords.

How can we tell which is which? There may be some delay before the network responds, so the first prompt to appear might not correspond with the first site we tried to open. Both Netscape and Internet Explorer show two pieces of information in the password prompt: (a) the site hostname, which is the same for both sites in this example, and (b) the authentication "realm", a string that the Web master can configure. If Alice and Bob have both left the realm at some default value, their prompts will be indistinguishable. (Or if Bob is nasty, he could decide to name his realm "Alice's Auctions"!)

Notice also that any other program running on the user's machine is free to bring up a window that looks exactly like one of these password prompts. With careful (or lucky) timing, this other program could fool the user into giving it a secret password.

Analysis: The problem of the two identical password prompts is due to a violation of the principle of **identifiability**. The password prompt is vulnerable to spoofing because there is no **trusted path**.

Solution: Figure 5 suggests a possible design that would solve both of these problems. We first introduce the rule that the operating system only allows applications to draw into rectangular frame buffers, which the system copies onto the screen. The system has control over the window borders and the desktop area. Then we change the Web browser so that it asks the operating system to request user authentication on its behalf. The system-generated password prompt is drawn with a special red striped border that no application could imitate, eliminating the possibility of spoofing. Red lines join the prompt window to the window of the requesting application, establishing an unmistakable association between the two.

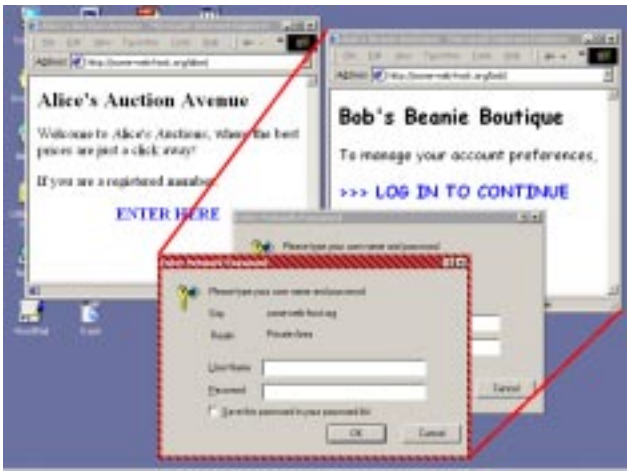


Figure 5. A possible solution to trusted path and identifiability issues.

Turning the password requester into an operating system function has the advantage that it would permit one to implement a challenge-response mechanism, to avoid trusting applications with passwords in the clear. It also facilitates the addition of operating system features for managing authentication in general.

4.8. Unix File Permissions

Problem: Alice, Bob, and Carol have accounts on a Unix system. The Unix file permission system assigns a single owner ID and group ID to each file. Separate flags give permission to read, write, or execute the file. There are three flags specifying these permissions for the file's owner, three for members of the file's group, and three for all other users. Groups can be created and edited only by the system administrator.

Alice has a file named "`secrets.txt`" that she would like to share with Bob. She trusts Bob with the secrets in this file, but doesn't want anyone else to see them. There are some groups defined for people in her office and people in Bob's office, but no group containing just her and Bob. What can Alice do?

The next week, Alice and Bob are working on the file together. Alice would like them both to be able to edit the file. She also wants to let her friend Carol read the file, but not to change it. Now what does she do?

```
unix% ls -l secrets.txt
-rw-------  1 alice    alice     8743 Jan 31 00:47 secrets.txt
unix% grep alicebob /etc/group
alicebob:x:100:alice,bob
unix% chgrp alicebob secrets.txt
unix% chmod g+r secrets.txt
unix% ls -l secrets.txt
-rw-r-----  1 alice    alicebob  8743 Jan 31 00:47 secrets.txt
unix%
```

Figure 6. Alice shares a secret file with Bob.

Analysis: In the first case, Alice (on her own) cannot share her file with only Bob. She must ask the system administrator to help her by creating a new group "alicebob" con
her file's group to "alicebob" and turns on "gro
access. Figure 6 shows what her Unix shell session might look like. In the second case, we must disappoint Alice. There is no way to achieve what she wants in the Unix filesystem, because each file has only one group ID.

Although the permission system does distinguish different files, users, and types of access, it is still insufficiently **expressive** for many common tasks, such as the one just described. It is impossible to share a file among a specific set of users unless a group containing that set of users already exists. The system administrator isn't likely to create a special group each time a user wants to share files with a new combination of people. Often the only available way to do any sharing is to give access to all users on the system, which is an unfortunate and excessive privacy risk.

Solution: Of course, it would be better to use a more flexible permission system. Ideally one would like to be able to give commands such as "grant Bob a
modify this file" and "grant Carol access to read this file". Figure 7 shows how Alice might solve her second problem in a hypothetical system with such a "`grant`" command. User and group names don't appear in the listings produced by "`ls -l`" because they aren't necessary; instead, the number in the first column is the number of outstanding grants.

```
unix% ls -l secrets.txt
  0     8743 Jan 31 00:47 secrets.txt
unix% grant bob +rw secrets.txt
unix% grant carol +r secrets.txt
unix% ls -l secrets.txt
  2     8743 Jan 31 00:47 secrets.txt

       . . . time passes . . .

unix% lsgrants secrets.txt
  1. granted +rw to bob (Jan 31 00:53)
  2. granted +r to carol (Jan 31 00:54)
unix% revoke 1 secrets.txt
unix% ls -l secrets.txt
  1     8743 Jan 31 00:47 secrets.txt
unix% lsgrants secrets.txt
  1. granted +r to carol (Jan 31 00:54)
unix%
```

Figure 7. Alice gives write access to Bob and read access to Carol. Later, she revokes Bob's access.

4.9. Java Applet Privileges in Netscape

Problem: Since version 3.0, Netscape Navigator has managed security for Java applets by allowing the user to grant and deny what the Netscape documentation alternately calls "privileges" or "capabilities". Before an applet is allowed to perform potentially dangerous operations, such as connecting to the network or accessing local files, it must first activate an associated privilege with an `enablePrivilege(...)` call to Netscape's SecurityManager. This usually causes a dialog box to appear asking the user to grant the privilege in question, as in Figure 8.

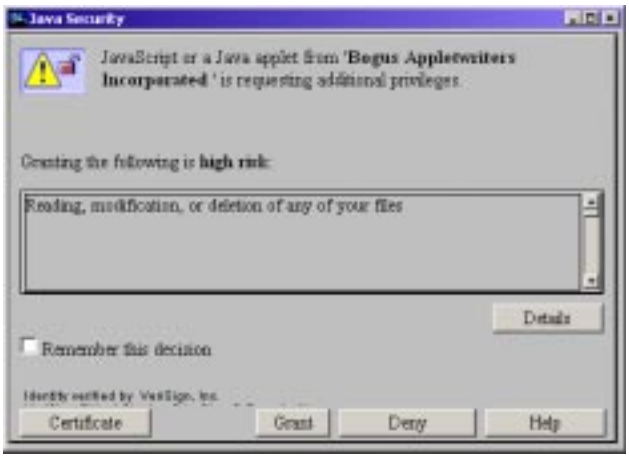


Figure 8. Applet requests a privilege by calling enablePrivilege("UniversalFileAccess").

The dialog box omits a lot of important information. What program is going to receive the privilege, and how long will the privilege last? If the user chooses "Remember this decision", exactly what decision will be recorded, and how long will it stay in effect? If the user grants or denies the privilege now, how is the decision reversed later? As it turns out, choosing "Grant" gives the privilege to *all* scripts and applets from a given source, from now until the end of the Netscape session, without any further permission from the user. If the "Remember this decision" box is checked, the privilege lasts indefinitely, and is automatically granted to all programs from this source in all future Netscape sessions. The dialog box is so vague that no user could possibly make a reasonable decision.

Further, the user can't be certain that the program is really from Bogus Appletwriters Incorporated, as the certification details are obscured at the bottom of the window. Even if the window is resized, the user interface toolkit rearranges the widgets in the window to match, so the text remains obscured. This could be considered merely a programming bug, but it should be noted as a subtle security consequence of cross-platform user interface design.

Suppose that the user wishes to change the privilege settings for programs from Bogus Appletwriters Incorporated. After some privileges have been granted, the interface for editing privileges might look like Figure 9. Each privilege can be in one of three lists: "Always" (privileges that are always granted), "For this session only" (privileges that are automatically granted only for the remainder of the session), or "Never" (privileges that are automatically denied).

If the user selects the "Reading information..." privilege and clicks "Delete", a confirmation dialog appears, as in Figure 10. The sentence says that the privilege will be deleted "for all applets and scripts from Bogus". Yet the actual effect is to delete this privilege from the list of privileges *automatically denied*: that is, it will become *possible to grant* the privilege in future.

Figure 9. Editing privileges.

Figure 10. Deleting a privilege.

The latest version of Netscape, version 6.2, is even worse in this situation. It merely presents a single question, shown in Figure 11, that provides absolutely no information about the privileges to be granted. (In case the user has any doubts about granting unknown privileges to unnamed entities for unspecified intervals of time, with no knowledge of how to revoke them, the "Yes" button is helpfully selected by default.)

Figure 11. Privilege prompt in Netscape 6.

Analysis: All of these dialog boxes violate the principle of **clarity** by being ambiguous and sometimes misleading. The last one, by selecting "Yes" as the default, also ignores the path of least resistance.

Solution: These dialog boxes should be redesigned so that all the relevant information is presented and the explanations are specific and clear. If the interface toolkit varies from platform to platform, the security dialogs should be carefully tested on each platform.

5. Conclusion

In an attempt to provide a reasonable starting point for talking about user interaction in secure systems, we have presented the actor-ability model and a set of design principles, and done our best to justify the applicability of these principles. Our greatest hope is that this paper will provoke further thinking and discussion about a user-centred approach to computer security, and eventually lead to the construction of computer systems that are safer, more reliable, and easier to use.

6. Acknowledgements

This paper builds directly on previous work done jointly with Miriam Walker [Walker99].

Many of the key insights in this paper come from Norm Hardy, Mark S. Miller, Chip Morningstar, Kragen Sitaker, Marc Stiegler, and Dean Tribble, who devoted much of their time to extensive discussions during which the set of design principles was developed.

We are very grateful to Morgan Ames, Verna Arts, Nikita Borisov, Jeff Dunmall, Tal Garfinkel, Marti Hearst, Johann Hibschman, Josh Levenberg, Lisa Megna, David Wagner, and David Waters for their comments and suggestions, which have greatly helped to improve this paper.

7. References

[Adams99] A. Adams and M. A. Sasse. Users Are Not The Enemy: Why users compromise security mechanisms and how to take remedial measures. In *Communications of the ACM*, December 1999, pp. 40-46.

[Bruce88] B. Bruce and D. Newman. Interacting Plans. In *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann, 1988, pp. 248-267.

[CA-1999-04] CERT Advisory CA-1999-04: Melissa Macro Virus. http://www.cert.org/advisories/CA-1999-04.html, 27 March 1999.

[CA-2000-04] CERT Advisory CA-2000-04: Love Letter Worm. http://www.cert.org/advisories/CA-2000-04.html, 4 May 2000.

[CapDesk] E and CapDesk: POLA for the Distributed Desktop. (see http://www.combex.com/tech/edesk.html)

[Dennett87] D. Dennett. *The Intentional Stance*. Cambridge, MA: MIT Press, 1987.

[E] ERights.org: Open Source Distributed Capabilities. (see http://www.erights.org/)

[Gibson77] J. J. Gibson. *The Ecological Approach to Visual Perception*. Boston: Houghton Mifflin, 1979, p. 127 (excerpt at http://www.alamut.com/notebooks/a/affordances.html).

[Garfinkel96] S. Garfinkel and G. Spafford. *Practical UNIX and Internet Security: Second Edition*. O'Reilly & Associates, Inc., 1996.

[Hardy85] N. Hardy. The KeyKOS Architecture. In *Operating Systems Review*, vol. 19, no. 4, October 1985, pp. 8-25.

[Hardy88] N. Hardy. The Confused Deputy. In *Operating Systems Review*, vol. 22, no. 4, October 1988, pp. 36-38.

[Holmström99] U. Holmström. User-centered design of secure software. Human factors in Telecommunications, May 1999, Copenhagen, Denmark.

[Jendricke00] U. Jendricke and D. Gerd tom Markotten. Usability meets Security: The Identity-Manager as your Personal Security Assistant for the Internet. In *Proceedings of the 16th Annual Computer Security Applications Conference*, December 2000.

[Karat89] C.-M. Karat. Iterative Usability Testing of a Security Application. In *Proceedings of the Human Factors Society 33rd Annual Meeting*, 1989.

[Miller00] M. S. Miller, C. Morningstar, and B. Frantz. Capability-Based Financial Instruments. In *Proceedings of the 4th Conference on Financial Cryptography*, 2000.

[Mosteller89] W. S. Mosteller and J. Ballas. Usability Analysis of Messages from a Security System. In *Proceedings of the Human Factors Society 33rd Annual Meeting*, 1989.

[Microsoft98] Microsoft Security Bulletin MS98-010: Information on the "Back Orifice" Program. http://www.microsoft.com/technet/security/bulletin/ms98-010.asp, 12 August 1998.

[Nielsen94] J. Nielsen. Enhancing the explanatory power of usability heuristics. In *Proceedings of the CHI '94 Conference*, ACM Press, 1994, pp. 152-158.

[Norman88] D. A. Norman. *The Psychology of Everyday Things*. New York: Basic Books Inc., 1988.

[Nass94] C. Nass, J. Steuer, and E. Tauber. Computers are Social Actors. In *Proceedings of the CHI '94 Conference*, ACM Press, 1994, pp. 72-78 (see http://cyborganic.com/People/jonathan/Academia/Papers/Web/casa-chi-94.html).

[Saltzer75] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, vol. 63, no. 9, September 1975, pp. 1278-1308 (see http://web.mit.edu/Saltzer/www/publications/protection/).

[Shapiro99] J. Shapiro, J. Smith, and D. Farber. EROS: A Fast Capability System. In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP '99)*, December 1999.

[Whitten99] A. Whitten and J. D. Tygar. Why Johnny can't encrypt. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

[Walker99] M. Walker and K.-P. Yee. Interaction Design for End-User Security, http://www.cs.berkeley.edu/~pingster/sec/desktop/.

[Wertheimer23] M. Wertheimer. Untersuchungen zur Lehre von der Gestalt II. In *Psychologische Forschung*, 4, pp. 301-350. Condensed translation published as "Laws of organization in perceptual forms", in W. D. Ellis, *A Sourcebook of Gestalt Psychology*, London: Routledge & Kegan Paul, 1938, pp. 71-88 (http://psychclassics.yorku.ca/Wertheimer/Forms/forms.htm).

[Zurko99] M. E. Zurko, R. Simon, and T. Sanfilippo. A User-Centered, Modular Authorization Service Built on an RBAC Foundation. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, May 1999, pp. 57-71.