# User Interaction Design for Secure Systems

Ka-Ping Yee

University of California, Berkeley
ping@zesty.ca

**Abstract.** The security of any system that is configured or operated by human beings depends on the information conveyed by the user interface, the decisions of the users, and the interpretation of their actions. This paper establishes some starting points for reasoning about security from a user-centred perspective: it proposes to model systems in terms of actors and actions, and introduces the concept of the subjective *actor-ability state*. Ten principles for secure interaction design are identified; examples of real-world problems illustrate and justify the principles.

## 1 Introduction

Security problems are often attributed to software errors such as buffer overruns, race conditions, or weak cryptosystems. This has focused much attention on the correctness of software implementations. However, correct use of software is just as important as correctness of the software itself. For example, there is nothing inherently incorrect about a program that deletes files. It is only when such a program deletes files against our wishes that we perceive a security violation.

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" [7]. Notice that this definition is necessarily dependent on the meaning of "you", which usually refers to the user. It is impossible to even describe security without addressing the user perspective.

Among the most spectacular of recent security problems are the e-mail attachment viruses. Many of these are good real-life examples of security violations in the absence of software errors: at no point during their propagation does any software behave differently than its programmers would expect. The e-mail client correctly decodes the attachment and the system correctly executes the virus when the user opens the attachment. Rather, the problem exists because the functionally correct behaviour is inconsistent with what the user would want.

This paper aims to make two main contributions: first, it presents a model to guide thinking about this type of issue; and second, it gives specific recommendations in the form of ten interaction design principles for secure systems.

Among many designers, there is the pervasive assumption that improving security necessarily degrades usability, and vice versa; the decision to favour one or the other is typically seen as a regrettable compromise. For example, commonly suggested security fixes involve having the computer ask for user confirmation, yet we are also warned against annoying the user by asking too

frequently [14]. In the end, these judgement calls are often made arbitrarily. A coherent interaction model will help designers resolve these dilemmas.

I take the apparently radical position that security and usability are not fundamentally at odds with each other. In fact, it should be clear upon reflection that the opposite makes more sense: a more secure system is more controllable, more reliable, and hence more usable; a more usable system reduces confusion and is thus more likely to be secure. Security advocates and usability advocates both want computers to correctly do what users want – no more and no less[1].

The results presented here come from discussing design challenges and user experiences at length with designers and users of software intended to be secure. After much debate and several iterations of refinement, we have tried to form a concise set of design principles that covers many of the important failure modes.

This paper is heavily abridged due to last-minute space restrictions; for a complete version with detailed case studies, **please see http://zesty.ca/sid/**.

## 2 Related Work

There seem to be relatively few development efforts in computer security [10] [12] [25] that have seriously emphasized user interaction issues. The Adage project [25], a user-centred authorization service, is probably the largest such effort to date. There have been several important usability studies of security applications [1] [13] [16] [24], all of which have shown the devastating impact that ignoring usability issues can have on the effectiveness of security measures. To my knowledge, this paper is the first attempt to propose a structured framework for design thinking and to suggest widely applicable guidelines for secure interaction design as opposed to studying a single application or mechanism.

Simultaneously addressing all of the principles presented here is admittedly a significant design challenge. Lest they seem too idealistic to be satisfiable, it is worth mentioning that there is an independently developed prototype of a secure desktop shell [3] that largely succeeds in satisfying most of these principles.

## 3 Design Principles

The following sections present a preliminary set of guidelines for secure interaction design. They are a snapshot of an ongoing process of refinement; applying them in practice will help to assess their completeness. Sufficiency cannot be proved, as it is impossible to guarantee the success of a user interface. So, our criterion for admitting something as a basic principle is that it be a *necessary*

[1] Often a dilemma stems from conflicts between what different *people* want. For example, some digital rights management efforts would make media content harder to use – but the resulting conflict is not one of security versus usability. It is actually a conflict between the desires of the users and the content providers. Balancing such conflicts is indeed an important problem, but outside of the scope of this paper. We will not address the design of systems that serve two masters, but understanding how to serve one master faithfully remains an important and necessary first step.

and *non-trivial* concern. I will argue that each principle is necessary by showing how its violation would likely yield a security vulnerability, and show that the principles are non-trivial by pointing out that violations exist in real software.

The *principle of least privilege* [21] is a starting point for our reasoning. It is better to think of our design principles in the context of least-privilege systems, rather than current systems not designed in a least-privilege style. A language-based security system, such as Java's "sandbox", is one kind of model in which one could hope to satisfy these principles. Other platforms designed around the least-privilege concept include the E language [5], KeyKOS [8], and EROS [22].

The design principles are listed here, with detailed explanations to follow. In the statement of these principles, the term "actor" is used to mean approximately "user or program", but this term will be explained more precisely below. The term "authority" refers to the ability to take a particular action.

*Path of Least Resistance.* The most natural way to do any task should also be the most 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 Authorization.* A user's authorities must only be provided to other actors as a result of an explicit user action that is understood to imply granting.

*Visibility.* The interface should allow the user to easily review any active actors and 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 give the user 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 is meant by the user. For the purpose of this discussion, the user is a person at a computer using some interface devices such as a keyboard, mouse, and display. We are concerned with the software system that is intended to serve and protect the interests of the user, which is called the *user agent*.

On a single-user system, the user agent is the operating system shell, through which the user interacts with an arena of entities such as files and programs. On

a multi-user system, other users use their own user agents to interact in the same arena. When the system is connected to the Internet, there is a new 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 a Web browser) now represents the user's interests in that larger arena of interacting entities. 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 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.

This distinction is explained mainly to avoid confusion among levels of user agents. We will not directly address communicating through multiple user agents here; we consider only one level at a time. The rest of this paper discusses the design of any user agent serving a user. The ten design principles can apply to all kinds of users – not just end users of applications, but also system administrators or 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.

**Principle of the Path of Least Resistance.** 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 more concentration to use a hammer safely than unsafely, for instance.) We all have to learn, by being told, by observing others, and often by making painful mistakes, which ways of doing things are safe. Sometimes, through the design of our tools, we can make it easier to do things safely. Most food processors have a switch that lets them run only when the lid is closed. On power drills, the key for opening the drill chuck is often attached to the power cord so that unplugging the drill is a natural prerequisite to changing the drill bit. In both cases, a bit of cleverness has turned a safety precaution into a natural part of the way work is done, rather than an easily forgotten extra step.

Most users do not spend all their time thinking about security; rather, they are concerned with accomplishing some useful task. It is human nature to be economical with the use of physical and mental effort, and to choose the "path of least resistance". This can cause the user to work against security measures, either unintentionally or intentionally. The primary consideration is to keep the user's motivations and the security goals aligned with each other.

There are three aspects to this. First, observe that the ultimate path of least resistance is for the user to do nothing. Therefore, the default settings for any software should be secure (this is Saltzer and Schroeder's principle of "fail-safe defaults" [21]). It is unreasonable to expect users to read the documentation 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. User behaviour is largely guided by perceived *affordances* (the visual and non-visual properties of the interface that suggest the available modes of interaction) [6] [19]. For example, if a button for a security function does not look clickable, the user might never notice that it is an available action.

Third, consider whether users will subvert security intentionally. If operating securely takes too much effort, users may decide to circumvent or ignore security measures while completely aware that they are doing so. There is a security risk in systems where the secure patterns of usage are inconvenient: each inconvenience increases the probability that the user will decide to operate 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 security goals might seem to oppose the desire to make things easy. However, these are truly in conflict less often than one might think. Tightening security has to do with having more specific information about the user's intent so it can be achieved safely. Often this information is already conveyed in the user's actions; it just needs to be applied consistently to improve security[2].

In the few remaining situations where it is absolutely necessary to introduce a new inconvenience for the sake of security, provide a payoff by making productive use of the extra information the user is asked to provide. For example, consider a multi-user system that requires a login procedure. Entering a password is an extra step that is necessary for security, but has nothing to do with the user's intended task. However, the login information can be used to personalize the experience – by providing a custom desktop, menu of favourite programs, and so on – to offset the added inconvenience. This helps keep users from trying to circumvent the login process (or choosing to use a system that doesn't have one).

### 3.2 Objects, Actors, and Actions

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

Some objects are inert: their behaviour is simple enough to be completely predicted using physical laws. For instance, if a cup is pushed off of a table, we expect it to fall to the ground. In Dennett's terms, our model adopts the *physical stance* [4] toward the cup. On a computer, one might consider a text file such an object. One can perform actions on the text file (say, copy or delete it) with simple consequences, but the file does not appear to take actions of its own.

Some objects have their own behaviours; we will call such objects *actors*, since they are capable of taking action. 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. Since we cannot predict exactly what an actor will do, we proceed by estimating reasonable bounds on its behaviour.

To a computer user, an application program is an actor. There are some expectations about what the program will do, and some limits on what it should be able to do, but no user could know exactly what program instruction is being executed at a given moment. Even though the operation of the program may be

[2] See the file browser example in the section on explicit authorization.

completely deterministic, we cannot take a physical stance toward it because it is too complex. Instead, we base our model on an understanding of the purpose for which it was designed – Dennett calls this taking the *design stance*.

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 behaviours in terms of their beliefs and intentions; that is, we adopt what Dennett calls the *intentional stance*.

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

Building models of actors is something we humans are very good at. Bruce and Newman [2] 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 – yet such complex modelling is a routine skill for young children. There is also evidence from computer-human interaction research that people perceive computers as "social actors" [20] even though machines do not actually possess human motivations. Both these reasons suggest that we indeed form our mental models of computers in terms of actors and actions. It is no coincidence that this is reminiscent of OOP, since the designers of Smalltalk also sought to match our mental models of the world [11].

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*.

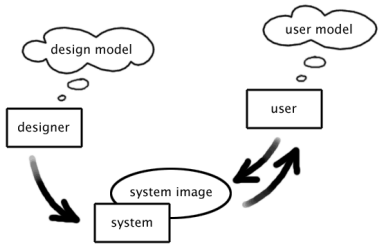


**Fig. 1.** The designer, the system, and the user (from [19]).

### 3.4 Aggregation

The actual working of a computer system is very complex and involves a great many small components and operations. To make the system comprehensible, the system image aggregates 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 connection, sending a request, and so on) it is presented as a single action.

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: users can perform *subjective aggregation* [15] on the file objects. Systems with end-user programming features, such as macros, allow the subjective aggregation of several actions into a single action.

**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 can perform action Y on object Z". The boundaries of the objects and actions are found by observing the system image, which conveys these boundaries through the methods it provides for identifying objects, communicating with actors, taking actions, and so on.

Here is an example to demonstrate the significance of choosing boundaries. Consider the idea that a secure operating system should let the user control the granting of authorities to programs. If a program spawns multiple processes, must the user separately grant authorities to each process? Or if a program relies on software modules or shared libraries, should the user have to separately control the authorities of every module? No: we resolve the dilemma by declaring that the boundaries between actors in the system image (which are also the boundaries of authority control) should be consistent with distinctions that the user 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.

Stated another way, the interface should distinguish objects and actions along boundaries that matter to the user. If the distinctions are too detailed, there is greater risk that users will overlap or leave out specifications. However, 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: Would the user ever want to manipulate this authority independently of another? To grant authority to this actor but not another? To permit access to this resource but not another?

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 components into useful conceptual units (that is, applications), such that reasoning about applications as individual actors holds valid. It follows that the system should enforce the corresponding boundaries: 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 some knowledge of actors and their abilities. As a starting point for talking about the user's conceptual state, let us consider a very simple model where the user knows about 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 $A_i$ do that would affect something I care about?" The knowledge of actors and abilities then consists of $\{\langle A_0, P_0\rangle, \langle A_1, P_1\rangle, \langle A_2, P_2\rangle, \ldots, \langle A_n, P_n\rangle\}$.

Since the user believes $P_0$ to be 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 actions that are actually possible, $P_0$ should be a subset of the user's real 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 $P_i$. To uphold this expectation, $P_i$ must be a superset of that actor's real abilities. If we write $R_i$ for actor $A_i$'s set of real abilities, our *no-surprise condition* can be summarized as follows:

$$P_0 \subseteq R_0 \qquad \text{and} \qquad P_i \supseteq R_i \text{ for } i > 0$$

**Principle of Explicit Authorization.** 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 changes in the actor-ability state. To maintain $P_i \supseteq R_i$, we require that only explicit user action can cause $R_i$ to come to exceed $P_i$.

Explicit authorization is perhaps the most basic requirement for controlling authority in any system, and 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. In current systems, applications often have complete access to the network and filesystem without ever having been explicitly granted these authorities.

At first glance, it may seem that this principle is in conflict with the principle of the path of least resistance. Must we constantly intercept the user with annoying security prompts to confirm every action? No – most of the time, the user already provides plenty of information in the course of performing tasks. The system must merely honour the manipulations of authority that are already being communicated. For example, if the user asks an application to open a file and makes a selection in 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. In many situations, combining designation with authority [9] yields an effective solution that improves both security and usability.

One can judge when explicit authorization is necessary on the basis of user expectations. For example, if there is a window that clearly belongs to an editor, one can expect the editor to draw in the window. However, it would certainly be unexpected for the editor to spontaneously delete the user's files. Just as it normally requires an explicit action for the user to delete files, so should it require explicit user action for another actor to acquire the ability to delete them.

The judgement of which authorizations should be explicit should be based on the potential consequences, not on the technical difficulty of the granting decision. *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 it 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 simpler terms.

**Principle of Visibility.** If the actor-ability state begins as a known quantity (some safe minimal set of authorities), and we are in control of each change in state, then in theory we know enough to ensure that our state remains accurate. However, there will be situations where one comes upon a new system in an unknown state. Moreover, it is unreasonable to expect a user to keep a perfect record of all grantings; human memory is fallible and limited in capacity. So we must enable users to update the actor-ability state in their heads at any time.

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 as essential for usability in general [18]. Likewise, visibility of authorities is necessary for users to understand the security implications of their actions. It makes sense to show the actor-ability state in terms of the granting actions that brought it about. Past granting actions having no effect on the current state (such as access given to applications that have terminated) need not be visible. It is helpful to identify authorities by inspection of either the holder or the accessible resource. Without visibility of authorities, any application could retain and use a granted authority undetected and indefinitely, once the user has forgotten about the granting action.

Windows and Unix systems typically run dozens of background processes. It should be emphasized that this principle does not require the interface to display all these processes. Processes like the kernel swap daemon are not part of the typical user's model and so should not be considered actors. Consequently, system behaviour should maintain consistency with a model where such processes are not actors: the system must strive to maintain the appearance that the swap daemon has no effect on the user's world of files and programs.

One of the most widely publicized examples of a harmful background process is the "Back Orifice" program released by Cult of the Dead Cow in 1998. This program *is* an actor since it can modify files and transmit them over the network without user initiation, and therefore should be visible in the interface. Although Microsoft denied [17] that there was a Windows security issue here, the fact that Windows allows Back Orifice to run invisibly is what makes it so dangerous.

**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 interface should allow the user to revoke granted authorities.

Another argument for facilitating revocation is the need to accommodate user error. It is inevitable that mistakes will be made; any well-designed system should help recover from them. In the context of granting authorities, error recovery amounts to revocation. One might intentionally grant an authority to an program and later discover that the program is untrustworthy; or one might inadvertently grant the wrong authority by mistake. In both cases, the granting decision should be reversible. Note that revocation prevents *further* abuse of an authority, but it is not always possible to undo damage caused by abuse of an authority while it was available. Thus, interfaces should avoid drawing an analogy between "revoke" and "undo"; "revoke" is better described as "desist".

**Principle of Expected Ability.** Whereas the preceding three principles deal with managing other actors' abilities, the perception of the user's own abilities can also have security consequences. Users sometimes make decisions based on the expectation of future abilities. There can be serious security consequences if these expectations are wrong. 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. Explicit authorization addresses one half of the no-surprise condition; this principle addresses the other half: $P_0 \subseteq R_0$.

For example, consider a system where granted authorities are usually revocable. If the user encounters an authority that cannot be revoked, the interface should make this clear, as it could affect the user's granting decisions.

### 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.

**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 user needs a trusted path to the interface components for handling permissions and authentication. Microsoft Windows, for example, provides a trusted path to its login window by requiring the user to press Ctrl-Alt-Del. These keys cause a non-maskable interrupt that can only be intercepted by the 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.

**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* (the same things should appear the same) and *discriminability* (different things 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. Even 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 Unicode, the issue is further complicated by characters that combine to form a single accented letter, which can cause different character sequences to be rendered identically on the screen.

It is not safe to assume that all actors will choose unique and consistent representations, so the continuity and discriminability of objects and actions are things that must be enforced by the system. This is the principle of identifiability.

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

If the language used to express policies 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.

For a good example of an expressiveness problem in real life, consider the Unix filesystem. Since each file can only be assigned to one group, it is impossible to share a file only for reading by one colleague and writing by another. Because the commands for setting permissions lack sufficient flexibility to express some kinds of sharing, users are sometimes forced to share files unsafely with everyone.

**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 correct software to enforce limits on each actor, but correctness of the implementation is irrelevant if the policy being enforced is not what the user intended. This can occur if the interface gives misleading, ambiguous, or incomplete information.

The interface must be clear not only about granting or revoking authorities; the consequences of any security-relevant decision, such as a decision to reveal sensitive information, should be clear. All the information needed to make a good decision should be accurate and available *before* the action is taken.

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 signify an exclusive selection of one option from many, while square checkboxes signify

individual yes-or-no decisions. An ellipsis at the end of a menu command indicates that additional options must be specified before an action takes place, whereas the absence of an ellipsis implies that an action will occur immediately.

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, we might rely on the user to correctly associate the text describing an authority with the button that controls it. Clarity can be evaluated in terms of the Gestalt principles of perceptual grouping [23], which suggest that visual associations are guided by proximity, closure, symmetry, figure-ground separation, continuation, and similarity.

### 3.7 Summary

In order to be able to use a system safely in a world of unreliable and adversarial software, a user must have confidence in all of the following statements:

- Things don't become unsafe all by themselves. (Explicit Authorization)
- 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 I can and cannot do within the system. (Expected Ability)
- I can distinguish the things that matter to me. (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 Conclusion

I have argued that consideration of human factors is essential for security, and that security and usability do not have to be in conflict. In an attempt to provide some foundations for talking about secure interaction design, I have presented the actor-ability model and a set of design principles. The model is supported by evidence from other research; the principles are supported by direct reasoning, by the model, and by examples of problems in real software. The principles are applied in case studies of interaction problems, with proposed solutions, available at http://zesty.ca/sid/. I hope this paper will provoke discussion about a user-centred approach to computer security, and lead to computer systems that are safer and more reliable – not only in theory, but also in practice.

## 5 Acknowledgements

This paper builds directly on previous work with Miriam Walker. Many of the insights in this paper come from Norm Hardy, Mark S. Miller, Chip Morningstar, Kragen Sitaker, Marc Stiegler, and Dean Tribble, who participated in the extensive discussions during which the design principles were developed.

Thanks also to Morgan Ames, Verna Arts, Nikita Borisov, Jeff Dunmall, Tal Garfinkel, Marti Hearst, Nadia Heninger, Johann Hibschman, Josh Levenberg, Lisa Megna, David Wagner, and David Waters for help reviewing this paper.

## References


1. A. Adams and M. A. Sasse. Users are Not the Enemy. In Communications of the ACM (Dec 1999), p. 40–46.
2. B. Bruce and D. Newman. Interacting Plans. In Readings in Distributed Artificial Intelligence. Morgan Kaufmann (1988), p. 248–267.
3. Combex. E and CapDesk: POLA for the Distributed Desktop. http://www.combex.com/tech/edesk.html.
4. D. Dennett. The Intentional Stance. MIT Press (1987).
5. ERights.org: Open Source Distributed Capabilities. http://www.erights.org/.
6. J. J. Gibson. The Ecological Approach to Visual Perception. Houghton Mifflin (1979), p. 127 (excerpt, http://www.alamut.com/notebooks/a/affordances.html).
7. S. Garfinkel and G. Spafford. Practical UNIX and Internet Security. O'Reilly (1996).
8. N. Hardy. The KeyKOS Architecture. In Operating Systems Review, 19(4)8–25.
9. N. Hardy. The Confused Deputy. In Operating Systems Review, 22(4)36–38.
10. U. Holmström. User-centered design of secure software. In Proceeedings of the 17th Symposium on Human Factors in Telecommunications (May 1999), Denmark.
11. D. Ingalls. Design Principles Behind Smalltalk. In BYTE Magazine (Aug 1981).
12. 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 (Dec 2000).
13. C.-M. Karat. Iterative Usability Testing of a Security Application. In Proceedings of the Human Factors Society 33rd Annual Meeting (1989).
14. K. Karvonen. Creating Trust. In Proceedings of the Fourth Nordic Workshop on Secure IT Systems (Nov 1999), p. 21–36.
15. M. S. Miller, C. Morningstar, and B. Frantz. Capability-Based Financial Instruments. In Proceedings of the 4th Conference on Financial Cryptography (2000).
16. 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).
17. Microsoft. Bulletin MS98-010: Information on the "Back Orifice" Program. http://www.microsoft.com/technet/security/bulletin/ms98-010.asp (Aug 1998).
18. J. Nielsen. Enhancing the explanatory power of usability heuristics. In Proceedings of the ACM CHI Conference (1994), p. 152–158.
19. D. A. Norman. The Psychology of Everyday Things. New York: Basic Books (1988).
20. C. Nass, J. Steuer, and E. Tauber. Computers are Social Actors. In Proceedings of the ACM CHI Conference (1994), p. 72–78 (see http://cyborganic.com/People/jonathan/Academia/Papers/Web/casa-chi-94.html).
21. J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In Proceedings of the IEEE, 63(9)1278–1308 (see http://web.mit.edu/Saltzer/www/publications/protection/).
22. J. Shapiro, J. Smith, and D. Farber. EROS: A Fast Capability System. In Proceedings of the 17th ACM Symposium on Op. Sys. Principles (Dec 1999).
23. M. Wertheimer. Untersuchungen zur Lehre von der Gestalt II. In Psychologische Forschung, 4, p. 301–350. Translation "Laws of organization in perceptual forms" in W. D. Ellis, A Sourcebook of Gestalt Psychology, Routledge & Kegan Paul (1938), p. 71–88 (see http://psychclassics.yorku.ca/Wertheimer/Forms/forms.htm).
24. A. Whitten and J. D. Tygar. Why Johnny can't encrypt. In Proceedings of the 8th USENIX Security Symposium (Aug 1999).
25. 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), p. 57–71.