User Interaction Design for Secure Systems

Ka-Ping Yee

`http://zesty.ca/sid/`
`ping@zesty.ca`

Abstract. The security of any computer system that is configured or operated by human beings critically 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 point of view: it proposes to model systems in terms of actors and actions, and introduces the concept of the subjective *actor-ability state*. Ten key principles for secure interaction design are identified; case studies illustrate and justify the principles, describing real-world problems and possible solutions. It is hoped 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 happens 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 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" [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 e-mail attachment viruses. Many of these are good real-life examples of security violations in the absence of software errors: at no point in their propagation does any application or system software behave differently than its programmers would expect. The e-mail client correctly displays the message and correctly decodes the attachment; the system correctly executes the virus program 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.

Many designers habitually assume 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. For example, a commonly suggested security fix is to have the computer ask for user confirmation, yet we are also often warned against annoying the user by asking too frequently [14]. In the end, these judgement calls are often made arbitrarily because there seems to be no good answer. A coherent model for secure user interaction can clarify the design process and help designers make these decisions consistently.

I take the apparently radical position that security and usability are not fundamentally at odds with each other. In fact, it should become clear upon reflection that the opposite makes more sense: a system that is more secure is more controllable, more reliable, and hence more usable; a more usable system reduces confusion and is thus more likely to be secure. In general, security advocates and usability advocates both want the computer to correctly do what the user wants – 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 distill the most productive lines of reasoning down to a concise set of design principles that covers many of the important and common failure modes.

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 ten of the design principles presented here is admittedly a significant design challenge. Lest they seem too idealistic to be satisfiable by a real system, it is worth mentioning that there is an independently developed working prototype of a secure desktop environment [3] that largely succeeds in satisfying most of the principles.

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

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. Completeness cannot be proved, as it is impossible to guarantee the success of a user interface. Our criterion for admitting something as a basic principle is that it should be a *valid* and *non-trivial* concern. I will argue that each principle is valid by showing how a violation of the principle would lead to a security vulnerability. Examples given here and the case studies given in the appendix show that the principles are non-trivial by pointing out violations in real software.

Saltzer and Schroeder's *principle of least privilege* [21] is a basic starting point for our reasoning. It may be difficult to imagine how to meet all these principles in popular operating systems, since most are not designed to work in a least-privilege style. The principles will make more sense when considered in the context of a system that can support least privilege. For instance, Java's "sandbox" is a language-based security model in which one might be able to better satisfy some of these principles. Platforms designed specifically for least privilege include the E scripting 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" just 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 we mean 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 we call the *user agent*.

On a single-user 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.

This distinction is mentioned here mainly to avoid confusion among levels of user agents. We will not directly address the issue of communicating through multiple user agents; we will consider the context of 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, 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.

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 much 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 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 easily forgotten extra 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 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 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. 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 [6] to refer to the properties of things that determine how they can be interacted with. Don Norman applied this concept to interaction design [19]. In human-computer interfaces, user behaviour is largely guided by perceived affordances. For example, suppose an icon of a lock can be clicked to request detailed security information. If the icon is not made to look clickable, 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 might seem to conflict with the desire to make things secure. However, these goals are truly in conflict less often than one might think. Making security tighter usually has to do with getting more specific information about the user's goal so it can be achieved more safely. Often this information is already conveyed in the user's actions; it just needs to be applied consistently to improve security[2].

There remain some 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 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 little 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, personal document folders, and so on – to offset the added

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

inconvenience. This helps to keep users 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

In order to productively interact with the world around them, people build mental models of how it works. These models enable them to make predictions about the consequences of their actions, so they can make useful decisions. In these models, 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 inert: the way they interact with other things is simple enough to be modelled with physical laws. For instance, if a cup is pushed off the edge of a table, we expect it to fall to the ground. In Dennett's terminology, our model adopts the *physical stance* [4] toward the cup. It is straightforward to work with such objects because we can predict precisely what they will do. On a computer, one might consider a text file an example of such an object. One can do things to the text file (say, copy it or delete it) that have simple, predictable 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 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, the user cannot take a physical stance toward it because it is too complex. Instead, the user must model the program based on our 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 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: while one is inside a locked house, for example, one has no need to model the intentions of any people outside the house because one can rely 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, as we have been learning how to do it all our lives. 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 significant evidence from computer-human interaction research that people perceive computers as "social actors" [20] 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. It is no coincidence that this is reminiscent of object-oriented programming, 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*.

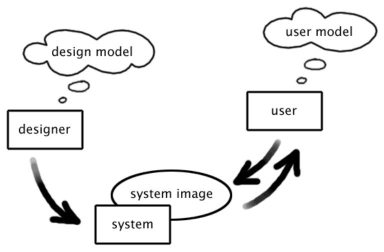


Fig. 1. The designer, the system, and the user (from [19]).

3.4 Aggregation

The actual working of a computer system is extremely intricate 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 [2] 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, the user can perform *subjective aggregation* [15] 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.

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 these boundaries. Consider the basic idea that a secure operating system should allow the user to control the granting of authorities to applications. If an application 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, 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 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: Would the user ever want to manipulate this authority independently of another? To grant an 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.

The Java security settings in Internet Explorer 5 demonstrate a lack of appropriate boundaries, to note one specific example. There is only one kind of setting for filesystem access: full access to all files is either granted or denied. This completely ignores the important boundaries between different file objects in the user model, making it impossible to offer a Java applet access to anything on the disk in any reasonably safe way.

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, we suggest 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 $\{\langle A_0, P_0\rangle, \langle A_1, P_1\rangle, \langle A_2, P_2\rangle, \ldots, \langle A_n, P_n\rangle\}$. We will call this subjective information the user's *actor-ability state*.

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:

$$\begin{aligned} P_0 &\subseteq R_0 && \text{and} \\ P_i &\supseteq R_i && \text{for } i > 0 \end{aligned}$$

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 any changes that would affect the actor-ability state. More precisely, since the user's actor-ability state is a set of *bounds* on each actor's abilities (rather than an enumeration of each specific ability), we require that only the user can cause R_i to come to exceed P_i, and that the user must do so by explicit action. This maintains the no-surprise condition.

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 global access to the network and filesystem without ever having been explicitly granted these authorities.

At first glance, it may seem that the principle of explicit authorization is in conflict with the principle of the path of least resistance. Does the principle of explicit authorization mean that we must constantly intercept the user with annoying security prompts to confirm every action? No – in fact, most of the time, extra confirmation is avoidable; the user already provides plenty of information in the course of performing the task. The system merely needs to 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 would require 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 what authorizations 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 are easier to understand.

Principle of Visibility. If the actor-ability state begins as a known quantity (say, with a safe minimal set of authorities, such as allowing programs 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.

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 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. Since 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 having no effect on the current state (such as access given to a program that has since terminated) need not be visible. It is helpful to be able to identify authorities by inspection of either the holder or the accessible resource. Without visibility of authorities, any application that gains an authority could retain and use the authority undetected and indefinitely, once the user has forgotten about the granting action.

Windows and Unix systems typically run dozens of background system processes. It should be emphasized that this principle does not require the interface to display all these processes. Processes like the window manager and the kernel swap daemon are not part of the typical user's conceptual model and therefore should not be considered actors. Consequently, what this principle *does* require is that the system behaviour 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. The Back Orifice 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 any Windows security issue here, the fact that Windows allows Back Orifice to run invisibly and exercise ungranted authorities is exactly 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 user should be allowed to revoke granted authorities; this is the principle of revocability.

Another 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 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 the abuse of an authority while it was available. Therefore, interfaces should avoid drawing an analogy between "revoke" and "undo"; instead, "revoke" is better described as "desist".

Principle of Expected Ability. Whereas the preceding three principles deal with managing perceptions of other actors' abilities, the user's perception of his or her own abilities 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. 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 interface must not give the user the false impression of an ability. Explicit authorization addresses one half of the no-surprise condition; this principle addresses the other half: $P_0 \subseteq R_0$.

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.

If the interface gives the impression that users can delete files when in fact they cannot, this might lead users to record secrets with the false expectation that they can later be destroyed. Or if users lack the authority to save files and the interface fails to indicate this, users might invest a lot of effort before discovering that all their work must be discarded.

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 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. This issue also needs to be addressed in any language system for running untrusted code, such as Java.

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. 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 Unicode, the issue is further complicated by characters that combine to form a single accented letter, as this means that different character sequences can be rendered identically on the screen.

As the 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.

Principle of Expressiveness. 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.

For a good example of an expressiveness problem in real life, consider the standard Unix filesystem. Since each file can only be assigned to one group, it is impossible to share a file for reading by one colleague and writing by another, without exposing the file to global access. The access control mechanism does not have sufficient flexibility to express this kind of sharing. Users who want to share files with multiple groups or with users not already in administrator-defined groups are forced to share files unsafely with the whole world.

Principle of Clarity. When the user is given control to manipulate authorities, we must ensure that the results reflect the user's intent. Although we may rely on software correctness to enforce limits on the authorities available to an actor, 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 needed to make a good decision should be 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 specified 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 [23] can be applied to evaluate and improve clarity:

– Proximity: items near each other belong together.
– Closure: line breaks and form discontinuities are filled in.

– Symmetry: symmetrically positioned and shaped objects belong together.
– Figure-ground segregation: small objects are seen as the foreground.
– Continuation: objects that follow a line or curve belong 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, a user needs to 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 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)

We have touched on a few examples of violations during the exposition of the principles; the appendix provides some more detailed case studies of real-world situations where these design principles are violated, and suggests solutions.

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 security problems in real software. 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 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.

A Case Studies

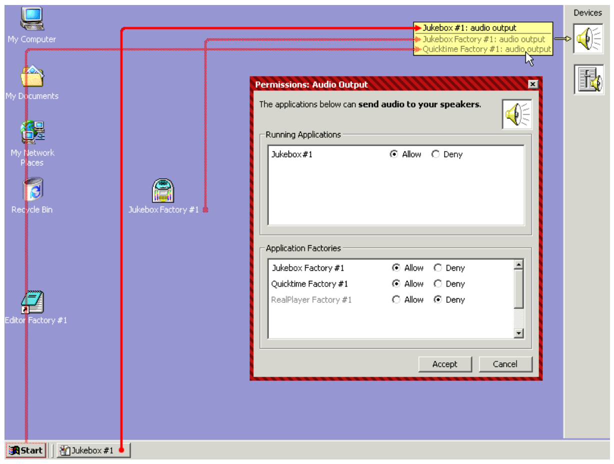
The following sections analyze some security problems that arise from usability issues in real-life situations, and show how our design principles apply.

A.1 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. The installation process is a complete mystery: there is no indication what resources are given to the new software, what global settings are modified, or how to restore the system to a stable state if installation fails. Configuration changes can leave the system in a state where other software or hardware no longer functions. Frequently the only available recourse is to try to guess what settings might be changed and write them down on paper in advance.

Analysis. Control over software and hardware components should not be handed over without the user's permission, as this violates the principle of *explicit authorization*. Just as important is the user's ability to inspect and revoke authorities so that the system can be restored to a working state. Current systems fail to satisfy the principles of *visibility* and *revocability*.

Solution. Installing a new component of an audio system is not that different from installing software. One might install a new speaker by connecting it to the turntable, allowing the turntable to employ the speaker to generate sound. At any time, one can revoke all access to the speaker by disconnecting all cables leading to it; then one can have complete confidence that the speaker is unaffected by the rest of the system, and one is free to take it and use it elsewhere. Ideally, adding and removing software components should approach this level of simplicity.



Jukebox #1: audio output
Jukebox Factory #1: audio output
Quicktime Factory #1: audio output

Permissions: Audio Output

The applications below can **send audio to your speakers.**

Running Applications

Jukebox #1	[✓ Allow] Deny

Application Factories

Jukebox Factory #1	[✓ Allow] Deny
Quicktime Factory #1	[✓ Allow] Deny
RealPlayer Factory #1	Allow [✓ Deny]

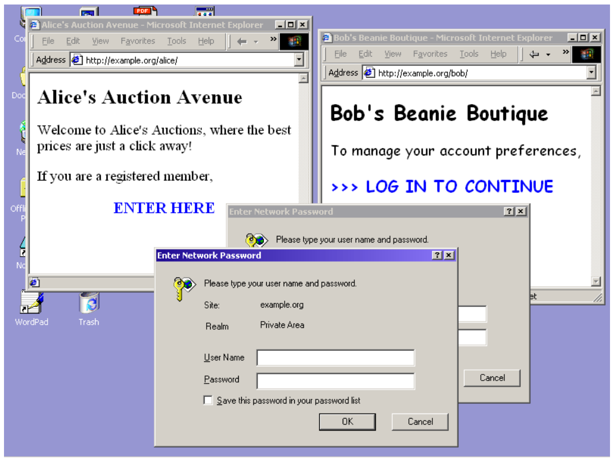
Fig. 2. A possible interface for revoking authorities.

Figure 2 is a possible interface design that might address these issues, based on the audio-system analogy. In this example, 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. The user has asked to inspect 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.

The general problem of software installation is complex; although a complete solution is outside the scope of this paper, this design example should help demonstrate that some progress is possible.

A.2 Website Password Prompts

Problem. Suppose that Alice and Bob both run Web sites requiring user authentication. Both use the same free hosting service at `example.org`. If we open two browser windows, one at each site, and attempt to enter the protected areas, two password prompts will appear, as in Fig. 3.



Alice's Auction Avenue - Microsoft Internet Explorer

http://example.org/alice/

Alice's Auction Avenue

Welcome to Alice's Auctions, where the best prices are just a click away!

If you are a registered member,

ENTER HERE

Bob's Beanie Boutique - Microsoft Internet Explorer

http://example.org/bob/

Bob's Beanie Boutique

To manage your account preferences,

>>> LOG IN TO CONTINUE

Enter Network Password

Please type your user name and password.

Enter Network Password

Please type your user name and password.

Site: example.org

Realm Private Area

User Name

Password

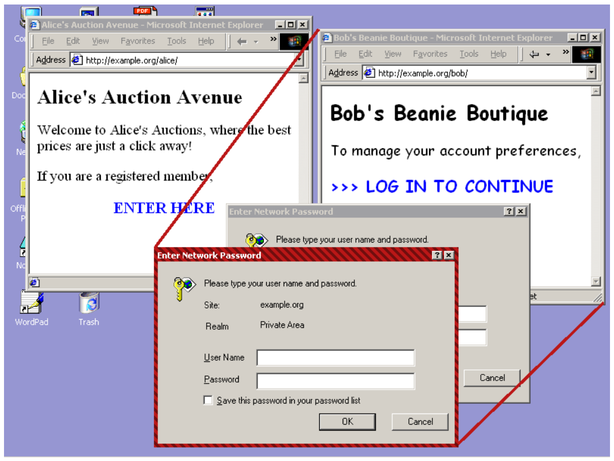
Save this password in your password list

Fig. 3. Two browser windows ask for passwords.

How can we tell which is which? Due to network delay, the first prompt to appear might not be the first site we tried to open. Both Netscape and Internet Explorer show two pieces of information in the prompts: (a) the hostname, which is the same for both sites, and (b) the authentication "realm", a string that the site operator 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 open a window that looks exactly like one of these prompts, luring the user into revealing a secret password.

Analysis. The problem of the identical prompts is a violation of the principle of *identifiability*. The prompt is vulnerable to spoofing by another program because there is no *trusted path*.



Alice's Auction Avenue

Welcome to Alice's Auctions, where the best prices are just a click away!

If you are a registered member,

Bob's Beanie Boutique

To manage your account preferences,

Please type your user name and password.

Site: example.org

Realm Private Area

Alice's Auction Avenue - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://example.org/alice/
ENTER HERE
Bob's Beanie Boutique - Microsoft Internet Explorer
Address http://example.org/bob/
>>> LOG IN TO CONTINUE
Enter Network Password
User Name
Password
Save this password in your password list
OK
Cancel
WordPad
Trash

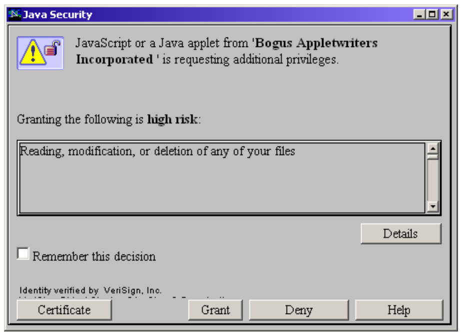
Fig. 4. A possible solution to trusted path and identifiability problems.

Solution. Figure 4 suggests a possible design that would solve both of these problems. We first introduce the rule that applications are only allowed to draw into rectangular frame buffers, which the system copies onto the screen. The system manages the window borders and the desktop area. Then we change the Web browser so it asks the operating system to request user authentication on its behalf. The system-generated password prompt is drawn with a red striped border that no application could imitate, eliminating the possibility of spoofing. The prompt could even be animated or faded in to demonstrate the system's exclusive ability to draw anywhere. Red lines join the prompt window to the window of the requesting application, establishing an unmistakable association.

A.3 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 are alternately called "privileges" or "capabilities" in the Netscape documentation. Before an applet is allowed to perform potentially dangerous operations, it must first activate an associated privilege with an `enablePrivilege(...)` call to the SecurityManager. This usually causes a dialog box to appear asking the user to grant the privilege in question, as in Fig. 5.

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, how long will it stay in effect, and how can the user reverse the decision later? It turns out that choosing "Grant" gives the privilege to *all* scripts and applets from a given source; if the "Remember this decision" box is checked, the privilege lasts


Java Security

JavaScript or a Java applet from '**Bogus Appletwriters Incorporated** ' is requesting additional privileges.

Granting the following is **high risk**:

Reading, modification, or deletion of any of your files

☐ Remember this decision

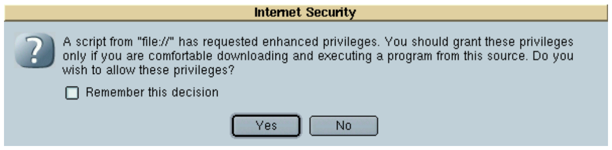
Identity verified by VeriSign, Inc.

Fig. 5. Applet requests a privilege using `enablePrivilege("UniversalFileAccess")`.

indefinitely, and is automatically granted in all future Netscape sessions. The dialog box is so vague that the user can't 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. When the window is resized, the UI toolkit rearranges the widgets to match, so the text remains obscured. This kind of bug is a subtle security consequence of cross-platform interface design.

The latest version of Netscape, version 6.2, is even worse in this situation. It always presents a single question, shown in Fig. 6, 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 unsecified intervals of time, with no knowledge of how to revoke them, the "Yes" button is helpfully selected by default.


Internet Security

A script from "file://" has requested enhanced privileges. You should grant these privileges only if you are comfortable downloading and executing a program from this source. Do you wish to allow these privileges?

☐ Remember this decision

Fig. 6. Privilege prompt in Netscape 6.

Analysis. Both of these dialog boxes violate the principle of *clarity* by being ambiguous and sometimes misleading. Both also fail to provide *appropriate boundaries* between distinct file objects or between distinct actors (that is, different Java applets). The second, by selecting "Yes" as the default, also flagrantly ignores the principle of 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 UI toolkit varies from platform to platform, the security dialogs should be carefully tested on each platform. The lack of boundaries between distinct actors can be addressed by fixing the prompting mechanism, but the lack of boundaries between files comes from a more fundamental limitation of Netscape's Java security model.

A.4 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 accepted, a malicious ActiveX control would have full access to the 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 may be 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 the assumption that users will always expend time and effort on security checks, nor on the assumption that all programs from trusted sources are infallible. By default, downloaded code should run with very 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.

A.5 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, according to CERT Advisory 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. The Love Letter virus did more damage than Melissa, destroying most of the image and music files on infected machines.

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 book.
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. The user adopts the physical stance [4] toward what appears to be an inert object, but the system turns that object into a new actor for which the design stance would be more appropriate. 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 authorization.* 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 authorization, and avoid giving out the authority to destroy files or send e-mail unless the user specifically authorizes this.