Always consider the users

Almost every software system in existence today interacts in one way or another with human beings. The users of a software system range from those in charge of fielding, configuring, and maintaining it operationally to those who actually use it for its intended purpose, the system's "end users."

The security stance of a software system is inextricably linked to what its users do with it. It is therefore very important that all security-related mechanisms are designed in a manner that makes it easy to deploy, configure, use, and update the system securely. Remember, security is not a feature that can simply be added to a software system, but rather a property emerging from how the system was built and is operated.

The way each user interacts with software is dictated not only by the design and implementation decisions of its creators but also by the cognitive abilities and cultural background of its users. Consequently, it is important that software designers and architects consider how the physical abilities, cultural biases, habits, and idiosyncrasies of the intended users of the system will impact its overall security stance. It is also a truism that during the life of any moderately useful system, a few users will discover capabilities that are outside the intentions of the system's designers and builders. Some of those capabilities may very well have significant security implications.

Usability and user experience considerations are often the most important factors ensuring that software systems operate in a secure manner. Designing systems that can be configured and used in a secure manner with easy-to-use, intuitive interfaces and sufficiently expressive, but not excessive, security controls is crucial.

However, it is dangerous to assume that every intended user of the system will be interested in security -- or will even be well-meaning. The challenge to designers and architects lies in creating designs that facilitate secure configuration and use by those interested in doing so, designs that motivate and incentivize secure use among those not particularly interested in software security, and designs that prevent or mitigate abuse from those who

intend to weaken or compromise the system.

Failing to address this design principle can lead to a number of problems

- Privilege escalation may result from a failure to implement an authorization model that is sufficiently tied to the authenticated entity (user) in all cases. Escalation failures may also occur when higher-privileged functions are not protected by the authorization model and where assumptions about inaccessibility are incorrect.
- A particular failure of appropriate authorization can allow a breach of the intended authorization and isolation between users such that one user may access another user's data.
- When designers don't "remember the user" in their software design, inadvertent disclosures by the user may take place. If it is difficult to understand the authorization model, or difficult to understand the configuration for visibility of data, then the user's data are likely to be unintentionally disclosed.
- Default configurations that are "open" (that is, default configurations that allow access to the system or data while the system is being configured or on the first run) assume that the first user is sophisticated enough to understand that other protections must be in place while the system is configured. Assumptions about the sophistication or security knowledge of users are bound to be incorrect some percentage of the time. This is particularly true at the startup and initialization of the system.
- If the security configuration is difficult or non-intuitive, the result will be an inability to configure the product to conform to the required security policy.
- Designers sometimes fail to account for the fact that authenticated and properly authorized users can also be attackers! This design error is a failure to distrust the user, resulting in authorized users having opportunities to misuse the system.
- When security is too hard to set up for a large population of the system's users, it will never be configured, or it will not be configured properly. This is especially dangerous where the system's defaults are "open" or insecure. For example, if there are too many clicks required for the user to get from the main page or screen to a security control panel, users are unlikely to persist through the labyrinth of clicks.
- Failure to consider the needs of programmers who must code to an API will cause the intended automation patterns to be missed. Programmers are a class of users who also require that the interface they consume be intuitive enough to guide them to correct usage patterns. Because a misunderstanding of an API occurs within the program that uses it, problems may not be readily apparent (appearing perhaps only obliquely, within log files of the ongoing activity), and the debugging of the problem difficult; this failure can be one of the most difficult to find and fix. Additionally, if the API must be changed, many if not all consumers of the API may be forced into further changes, thus spreading the original failure throughout the ecosystem.
- Failure to consider the possibility of "collateral damage" that can occur from included or embedded software or data in the user interface may cause an inadvertent or unintentional leaks of personal data. Consider the

capture of a bystander in a personal photo taken in a public place. Even if that passerby is not a user of software, the bystander's privacy may be compromised if that image is posted online later.

- Failure to consider the user's data during setup, use, and revocation/termination may cause unintended data to be gathered and stored against the users' wishes, or may hold onto data that should've been removed completely after the user has stopped using the service and closed his or her account. For example, when a user decides to stop using the system, is the private data easy for the user to destroy?
- Failure to consider the many different classes of users (blind users, language proficiency, children, people with different mental capabilities, etc.) will exclude those classes of users from the software — or, alternatively, make the software too difficult to use effectively. Most importantly, when designing the security of the system, failure to consider how security is set up and used from the perspective of users with different capabilities and understandings typically causes those users to set up and make inappropriate use of the software's security.

Stepping back, our biggest recommendation is the following: Always consider the users, and any other stakeholders, in the design and evaluation of systems. There are numerous factors to consider, and there are often trade-offs; for example, improving the system with respect to one user value (such as privacy or usability) can negatively affect another user value (like ease of accessing the relevant information).

In addition to the general recommendations given above, there are numerous artifacts designers can consider in order to address specific problems mentioned earlier. The decision whether to implement these specific recommendations will, however, depend on the system in question. For example, in some cases we recommend not putting security-relevant decisions in the hands of all users, as they may not possess the knowledge or context to evaluate those decisions. Similarly, because users may not know how to explore or choose between a variety of options, we recommend making the easiest and most common usage scenario also secure — a notion often referred to as "secure by default." When users do desire to change security settings, we suggest making it as easy as possible for them to find the relevant settings.

Often there is value in allowing users to test different security and privacy settings and see the results in order to understand the impact of the changes (e.g., on social networks, good interfaces allow users to see their privacy-settings changes from the perspective of other users).

On the other hand, it might be preferable not to give the user a choice at all; or example if a default secure choice does not have any material disadvantage over any other; if the choice is in a domain that the user is unlikely to be able to reason about; or if one user's choice may significantly affect the system's or the other user's state, including security.

Designers must also consider the implications of user fatigue (for example, the implications of having a user click "OK" every time an application needs a specific permission) and try to design a system that avoids user fatigue

while also providing the desired level of security and privacy to the user.

The field of user-focused security is rich with tensions. As a trivial example, so-called "secure" password selection strategies are also well known to lead to passwords that are hard for users to remember. A more complex example of these inherent tensions would be the need to make security simple enough for typical users while also giving sophisticated or administrative users the control that they require. We encourage designers to also consider other resources on designing security systems with stakeholders in mind.

By fully considering all the relevant stakeholders, designers have the opportunity to create systems that are both secure and usable, systems that will see adoption, and systems that will be compatible with the values of users and other people impacted by them.