Be flexible when considering future changes to objects and actors

- Introduction, Mission Statement, Preamble
- Earn or give, but never assume, trust
- Use an authentication mechanism that cannot be bypassed or tampered with
- Authorize after you authenticate
- Strictly separate data and control instructions, and never process control instructions received from untrusted sources
- Define an approach that ensures all data are explicitly validated
- Use cryptography correctly
- Identify sensitive data and how they should be handled

Software security must be designed for change, rather than being fragile, brittle, and static. During the design and development processes, the goal is to meet a set of functional and security requirements. However, software, the environments running software, and threats and attacks against software all change over time. Even when security is considered during design, or a framework being used was built correctly to permit runtime changes in a controlled and secure manner, designers still need to consider the security implications of future changes to objects and actors.

Designers need to understand how change influences security considerations under many circumstances. There will be changes at runtime, in the form of configuration changes, enabling and disabling of features, and sometimes dynamic loading of objects. The need for security consideration will appear during testing, since all possible variations of states will need to be verified to guarantee that they uphold the security posture of the system (among, of course, other tested behavior). There will be changes at deployment when permissions, access control and other security-related activities and decisions need to take place. The addition of continuous integration processes creates a requirement for security flexibility, as changes to systems are pushed automatically and at ever shorter periodicity.

Meanwhile, entropy increases in every way possible. Threats change over time. Embedded components (that is, components that are not easily reachable) will inevitably be found to be vulnerable to attacks, researchers will discover new ways to break into systems, and proprietary code will reveal itself to contain vulnerabilities. Any deployed system can eventually come under attack and potentially be compromised. And, because threats change over time, even a deployed system that has resisted attacks for a long time may eventually succumb to an attack and be compromised.

Like threats, the environment and conditions under which the system exists will also change. It is a different

Always consider the users
Understand how integrating external components changes your attack surface
Be flexible when considering future changes to objects and actors
Get Involved

proposition to maintain security for a system with 10 users than 10 million users -- not at all a simple matter of linear scale. A system that works well in a given configuration might find itself exposed to new threats by virtue of changes to that environment; for example, the addition of a mobile interface to a legacy system.

For these reasons, secure design keeps flexibility in mind.

Design for secure updates. It is easier to upgrade small pieces of a system than huge blobs. Doing so ensures that the security implications of the upgrade are well understood and controlled. For example, a database engine upgrade may involve new access control defaults or rewrites of the controls such that previously tight permissions loosen, or create new default users that need to be disabled. If the update happens with the same change operation performed on the web server, the amount of change and adjustment to a dynamic, already-configured system may be overwhelming to track and assure.

Have the system being upgraded verify the integrity and provenance of upgrade packages; make use of code signing and signed manifests to ensure that the system only consumes patches and updates of trusted origin. This is a non-trivial design consideration, as there are many details in process and implementation that may break if poorly thought out beforehand. Finally, consider the maintenance burden placed on administrative personnel. As complexity increases, there is an increasing likelihood of making mistakes.

Design for security properties changing over time; for example, when code is updated. If the system ran in a small environment yesterday, and local users and password storage were sufficient, tomorrow the system may be changed to make use of an alternate identity management solution. In that case, the migration of previous users (and/or the correct coexistence of the local and remote users) would need to happen in a way that does not compromise security; for example, there should be consideration of user ID collisions such as when a remote and a local user have the same username.

Design with the ability to isolate or toggle functionality. It should be possible to turn off compromised parts of the system, or to turn on performance-affecting mitigations, should the need arise. Not every vulnerability identified can be readily mitigated within a safe time period, and mission-critical systems cannot simply be taken offline until their vulnerabilities are addressed. For example, in certain environments a stateful firewall may impact performance overall, and so it is turned off — until a vulnerability that may be stopped by turning it on is identified, in which case it becomes worthwhile to bear the performance cost by turning the firewall on until a proper patch can be developed, tested, and applied.

Design for changes to objects intended to be kept secret. History has shown us that secrets such as encryption keys and passwords get compromised. Keeping secrets safe is a hard problem, and one should be prepared to have secrets replaced at any time and at all levels of the system. This includes several aspects:

- A secure way for users to change their own passwords, including disallowing the change until the old password has been successfully presented by the user.

- Carefully considering any kind of "password recovery" mechanism. It is better to give the forgetful user a way to reset their password after verification via a parallel mechanism (like email) than to provide the password in clear text, which can be subverted or compromised in any number of ways.
- A secure and efficient way to replace certificates, SSH keys, and other keys or authentication material that systems use, providing clear and explicit logs of those events (without including the secrets in any form!) in a forensically verifiable way (for example, external log servers and checksums).
- Understanding how the key change affects data stored at rest. For example, if data are encrypted on a file system or in a database and an administrator needs to change the encryption key, is it better to decrypt all data using the current key and re-encrypt that data with the new key, or to maintain versions of encrypted data and encryption keys?

Design for changes in the security properties of components beyond your control. Tech marches on. A cipher that was considered secure yesterday may be found to be less secure today, either by the discovery of an active attack against it or by improvements in hardware and software able to defeat that security control. In the same way, an external component's security properties or related characteristics may change over time, as when an Open Source project is abandoned and its code not actively maintained, or when its license changes, forcing users to abandon it.

In these cases it is important to design "agility," the capability to change layers and algorithms as needed, into the system. Good examples include Java's capability to change crypto providers without recompilation of classes, and Apache's capability of specifying a list of ciphers it is willing to negotiate with a client. Many hours of development and much grief over security flaws have been avoided due to these capabilities. Good design allows for intermediate layers of abstraction between code and imported external APIs, so that developers can change components providing needed functionality without changing much of the code.

Design for changes to entitlements. Systems are sometimes designed in which support staffers have privileged access to certain parts of the system in order to perform their job. However, the support staff's access to various system components likely changes over time. Individuals leave the organization, job functions change, they go on extended leaves or sabbaticals, system functionality changes, and so on. The system must have a way to revoke access to areas when a user no longer has a need to access them. This revocation of access should be part of an existing auditing mechanism in which access to critical system components is regularly reviewed to confirm that those individuals with access still require that level of access.