Use an authentication mechanism that cannot be bypassed or tampered with

Authentication is the act of validating an entity's identity. One goal of a secure design is to prevent an entity (user, attacker, or in general a "principal") from gaining access to a system or service without first authenticating. Once a user has been authenticated, a securely designed system should also prevent that user from changing identity without re-authentication. Authentication techniques require one or more factors such as: something you know (e.g., a password), something you are (e.g., biometrics such as fingerprints), or something you have (e.g., a smartphone). Multi-factor (sometimes referred to as N-factor) authentication refers to the technique of requiring multiple distinct factors to prove your identity. Authentication via a cookie stored on a browser client may be sufficient for some resources; stronger forms of authentication (e.g., a two-factor method) should be used for more sensitive functions, such as resetting a password. In general, a system should consider the strength of the authentication a user has provided before taking action. Note also that authentication encompasses more than just human-computer interaction; often, in large distributed systems, machines (and/or programs running on those machines) authenticate themselves to other machines.

The ability to bypass an authentication mechanism can result in an unauthorized entity having access to a system or service that it shouldn't. For example, a system that has an authentication mechanism, but allows a user to access the service by navigating directly to an "obscure" URL (i.e., a URL that is not directly linked to in a user interface, or that is simply otherwise "unknown" because a developer has not widely published it) within the service without also requiring an authentication credential, is vulnerable to authentication bypass.

The use of authentication techniques that don't fall into the category of something you know, something you are, or something you have may also allow users to access a system or service they shouldn't. System designers should beware of authentication techniques that depend on assumptions about sole possession of resources that may actually be shared. For example, authentication mechanisms that identify a user by their IP address wouldn't

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

be useful if the addresses were shared among different users at different times; for instance, via an address-sharing protocol such as DHCP.

Even when IP addresses are tied to particular devices, authentication based on device addresses is not a substitute for user authentication, as IP addresses can be spoofed and are not necessarily associated with specific users for a long time. As another concrete illustration, authentication mechanisms that rely on a computer's MAC address, which can easily be changed or spoofed, can result in unauthorized access if the device assumed to be identified with that individual is lost or stolen.

Typically, the act of authentication results in the creation of a token, capability (as often referred to in operating systems literature), or ticket representing a principal that is used throughout the system or service. If such tokens (or credentials) are deterministically derived from easy-to-obtain information, such as a user name, then it becomes possible to forge identities, allowing users to impersonate other users.

Credentials must not be easy to forge. Upon successful authentication, the user may be provided with an authentication credential, token, or ticket, which can be provided back to the system so that the user does not need to be re-authenticated for every request or transaction made via the system. At the same time, if it is possible for an attacker to forge the authentication credential, token, or ticket, the attacker can bypass the authentication mechanism. System designers can re-use time-tested authentication mechanisms such as Kerberos instead of building a new one. Alternatively, system designers are encouraged to use cryptography correctly (see the corresponding section later in this document) in constructing authentication credentials, tokens, and tickets.

If an authentication system does not limit the lifetime of an authentication interaction, then it may inadvertently grant access to a user to whom it should not. For example, imagine a user who logs into a public terminal and then walks away without logging out (which should terminate the session). A second user using the public terminal might now be able to use the system or service as the first user. A properly designed authentication system may automatically log the user out after a period of inactivity.

Authentication system designs should automatically provide a mechanism requiring re-authentication after a period of inactivity or prior to critical operations. As an example, upon receiving a transaction request to conduct certain sensitive actions such as changing a password, or transferring funds to another financial institution, a system could ask the user to re-enter their existing password again to confirm their transaction request, even though the user may already be authenticated.

The design of a system's re-authentication scheme, and when and how often to ask a user to re-enter their password, needs to be mindful of not only security, but also usability and convenience. Asking users to frequently re-enter their password can be damaging to security, as it trains people's muscle memory to enter their password every time they see a prompt and sets them up as easy phishing targets.

By far the most common authentication mechanism remains the password. Using passwords requires that the

system or service have a mechanism to associate a given password with a particular user. If this information is not properly stored, it may be possible for agents other than the user to obtain access to them. Storing such information securely is non-trivial, and the reader is referred to the use of an applied cryptography expert as noted in the *using cryptography correctly* section for guidance. Just as it is advisable to re-use tried and tested cryptographic algorithms, it is also advisable to re-use already built and tested password management systems instead of building new ones.

It's preferable to have a single method, component, or system responsible for authenticating users. Such a single mechanism can serve as a logical "choke point" that cannot be bypassed. Much as in code reuse, once a single mechanism has been determined to be correct, it makes sense to leverage it for all authentication.

To summarize, authentication mechanisms are critical to secure designs. They can be susceptible to various forms of tampering and may potentially be bypassed if not designed correctly. We recommend that a single authentication mechanism leverage one or more factors as per an application's requirements, that it serve as a "choke point" to avoid potential bypass, and that authentication credentials have limited lifetimes, be unforgeable, and be stored so that if the stored form is stolen, they cannot easily be used by the thief to pose as legitimate users.