Earn or give, but never assume, trust

Software systems comprising more than just a single monolithic component rely on the composition and cooperation of two or more software tiers or components to successfully accomplish their purpose. These designs often depend on the correct functioning of the existing parts. They will be inherently insecure if any of those parts are run in a potentially hostile environment, such as a user's desktop computer, an unmanaged device, or a runtime or sandbox that can be tampered with by an attacker.

Offloading security functions from server to client exposes those functions to a much less trustworthy environment, which is one of the most common causes of security failures predicated on misplaced trust.

Designs that place authorization, access control, enforcement of security policy, or embedded sensitive data in client software thinking that it won't be discovered, modified, or exposed by clever users or malicious attackers are inherently weak. Such designs will often lead to compromises.

Classic examples of software where trust is misplaced include a web browser or a thick-client application, but there are many more examples of client software. They include applications running on a mobile device, or embedded software that might be found in modern automobiles, pacemakers, gaming systems, or home appliances. Even calls into your APIs from business partners could be considered client software in some sense.

When untrusted clients send data to your system or perform a computation on its behalf, the data sent must be assumed to be compromised until proven otherwise. In some cases you may be able to guarantee that the client is, indeed, who it attests it is, or that the business logic it contains has not been altered or circumvented, or that external factors have not influenced the integrity of the computations it performed. But these situations are not the rule, and these underlying assumptions can change when new vulnerabilities are discovered. It is safer in the long run to design a software system under the assumption that components running on any platform whose

integrity can't be attested are inherently not trustable, and are therefore unsuitable for performing security sensitive tasks.

If, nonetheless, security operations must be offloaded to components running on an untrusted platform, the design should impose extreme caution on how the computation and its output are treated.

Common weaknesses related to client trust reside in various parts of the system, but tend to share a sensibility. A designer might (incorrectly) assume that server APIs will always be called in the same order every time. He might believe that the user interface is always able to restrict what the user is able to send to the server. He could try to build the business logic solely on the client side, or attempt to actually store a secret in the client. And, of course, a designer can run into danger by thinking that *any* intellectual property (IP) sent to the client can be protected through technical means.

Though security-aware development strategies cannot eliminate all these problems (or even resolve conflicts in goals for the software being developed), there are useful ways to minimize the potential risks. For example, some organizations will claim a real business need to store intellectual property or other sensitive material on the client. The first consideration is to confirm that sensitive material really does need to be stored on the client. When it truly is necessary to do so, various binary protection mechanisms can delay the leaking of sensitive material. Possible techniques to consider include obfuscation or anti-debugging (although the strength of these protections vary widely, so designers should understand the level of protection actually achieved with each tool or technique). Subject matter experts should be consulted if the system requires a client component with a level of protection that cannot be trivially compromised.

If IP or sensitive material must be stored or sent to the client, the system should be designed to be able to cope with potential compromise. For instance, the same shared secret or other cryptographic material shouldn't be used on all the clients. Make the validity of what is offloaded to the client limited in time, set expiration dates for data stored in the client, watermark IP, and double-check client computations that are security sensitive. On a related note, design your system to work in a limited fashion even when one or many clients have been completely compromised.

Finally, make sure all data received from an untrusted client are properly validated before processing. Follow the guidance described in the "*Define an approach that ensures all data are explicitly validated*" section above.

When designing your systems, be sure to consider the context where code will be executed, where data will go, and where data entering your system comes from. Failing to consider these things will expose you to vulnerabilities associated with trusting components that have not earned that trust.

IEEE Cybersecurity Initiative