SANS SECURITY AWARENESS

In this module, we focus on the second type of Sensitive Data Exposure; failure to ensure that application level information is properly protected while transmitted over the network, typically referred to as insufficient transport layer protection or TLS. This aspect of sensitive data exposure focuses on how data is protected while in motion and the need for encryption of application data when in transit. Consider these scenarios:

When a user reaches a point where it becomes necessary to authenticate, the application will likely prompt the user for login credentials. While the user's password resides in a database table as a hash, the password that is entered on the login form will be sent in clear text unless the application data is protected using an encryption technology like TLS. Failure to use TLS would allow an attacker who can observe the authentication, perhaps through a proxy or a sniffer, to steal the credentials and use them to impersonate the user.

Is it enough to encrypt the authentication of the user? No and here's why. At some point, a session is established and is tracked using a session token. This session token is generated and associated with the user when authentication takes place. From that point on, every time the user interacts with the application, the session token is used to re-authenticate every request. If communication with the application is not properly secured using TLS, an attacker can potentially intercept a copy of the session ID and again, impersonate the user.

For these reasons, we recommend that when encrypting communications between a user and the application, continue encrypting everything until the session is completed and the user logs out. We also recommend that whenever the state of the session changes, the current session token is destroyed and a new one created. Today, as the majority of web communication is always run over TLS, using TLS is considered good practice.

It is possible to write an application in such a way that encryption is always enforced. While we would still have a normal port 80 HTTP listener running, it's primary job is to let browsers know that they should be communicating over

HTTPS. This can be accomplished using a simple redirect, but there is a potential security issue. What if an attacker manages to trick a certificate authority into issuing him a valid certificate for someone else's site? In this case, using TLS is not enough, since the browser will view the certificate as valid. Here's what you can do to mitigate the threat.

The first part of the solution is to use the HTTP Strict Transport Security, or HSTS, header. This header, which is "Strict-Transport-Security," allows us to define a cache time in seconds and a set of domains for which the browser should only use HTTPS for communication. Considering the example of the attacker who has a fraudulent certificate, this clearly is not the complete solution.

The second part of the solution is Certificate Pinning, also known as Public Key Pinning, which uses the Public-Key-Pins HTTP header. Using this header, the application can inform the browser which hashes are the only valid hashes that should be accepted on a certificate on a website claiming to be the real application. If you are thinking that this will only work if the user goes to the real site before he goes to the attacker's fraudulent site, you are partially correct. Because an attacker could also attempt to pin the browser to the incorrect certificate, browser developers all provide public interfaces that can be used to register pinned certificate data that is used to pre-populate the browser. In this way, leveraging both strict transport security to require TLS for all web communication with public key pinning, creates a reasonable level of assurance that users will communicate with the real site rather than the attacker's site.

Still, there are many pre-existing applications that transition between HTTP and HTTPS. Remember, that whenever the state of the application changes, the session ID should be replaced. This includes both transitioning from HTTP to HTTPS, and from un-authenticated to authenticated. Yet, many frameworks and applications will issue a session token to the user before he has authenticated. So, once authentication occurs, the old session token should be destroyed and a new one issued. Otherwise, there is a Session Fixation issue, which means that the session was fixed prior to authentication or prior to encrypting. This is especially problematic if the initial session token was sent over an insecure channel. If the session token were intercepted at this point and it is not subsequently changed, it could be used by the attacker to impersonate the user's session.

Returning to the effective use of encryption to prevent sensitive data exposure, we have to consider user perception.

Imagine that a user has been using some sort of online shopping application. After

browsing around and adding several items to his shopping cart, he decides to checkout. On the check-out page, the application asks for him to either authenticate using an existing account or to fill in shipping and credit card information to complete his transaction. So far everything is as expected, but what if the user notices that this page, that is asking for sensitive information, was not sent to him using TLS? How can he tell? Simply by noticing that lock at the bottom of the window is missing.
If you were the user, would you be reluctant to submit your data to the application?

What's interesting about this example is that it is entirely possible that, when the submit link is clicked, the form will be submitted over TLS. If the user were to view the source code of the page he would see that the action in the form is set to an HTTPS reference. However, a user should never have to look at your source code to figure out if your application handles data securely.
What's the point? User perception tells us that any time our app asks for sensitive information, even the request, it should be sent over TLS.
A related user perception issue can occur with Mixed Frames or Mixed Content. Imagine the same app, but this time, when the application prompts the user for sensitive input, the page making that request is sent with TLS. What happens if an element within that page contains an absolute URL reference to an image or some other element and that absolute URL specifies HTTP rather than HTTPS? Again, the lock will not appear. In a sense, this is worse than the last example because the page was sent with TLS, but there's no easy way for the user to know this. Some browsers will even prompt the user with a warning, stating that all of the elements were not sent securely. The best solution to keep consumer confidence in the application is ensuring that all elements on the page are sent using TLS.
Issues related to sensitive data exposures can be addressed through properly encrypting data while in transit. We can force all secure content only to be delivered over TLS by using strict transport security and public key pinning; these are things that we can control within the application code or simple configuration of the web service. Other issues, such as mixed content or mixed frames, are user experience issues caused by developer choices or errors, and are typically easily resolved. Awareness of these issues should allow us to better explain and define the deployment requirements. Proper use of transport layer encryption controls goes a long way to preventing the exposure of sensitive data!