Problem Set 1

Joe Puccio

September 10, 2015

Prelude

So this all really relies on the fact that you have some certificates embedded in immutable hardware on devices, like the TPM, which are the certificates of the root CA's which are unconditionally trusted. These root CA's keep, and I believe sign, correspondences between names (e.g. "Apple Inc.") with company's public keys.

We start by creating a private key and public certificate, which will act as our own root CA. This root CA we create will be always trusted. We will use it to sign certificates for example websites. The private key of our root CA is also encrypted on disk, as plain-text is the work of the devil.

As an extra measure of security, we are asked to encrypt the private key we create for our example company using AES-128 encryption, relying on a password that we come up with.

Next, we create a certificate signing request by the example company to our previously created root CA. This certificate signing request contains our example company's public key, because our root CA will essentially be signing the example company's public key with their own private key, which will allow third parties who trust the root CA to trust that they are actually interacting with the example company, by checking the example company's certificate against the root CA's public key. Before generating the certificate for the example company, the root CA should really verify the identity of who sent the CSR, perhaps by requesting that the server put some certain text file with specified contents on a public directory that the root CA can view?

3.3

After signing the example company's certificate with the root CA, and adding the root CA's certificate to Firefox as a trusted CA, navigating to the domain does not result in an error. Displayed are all of the possible ciphers supported by the browser and the server for encrypting traffic, that is, securing a channel.

After modifying a byte of the certificate, loading the webpage for the first time took quite some time (a few seconds), compared to subsequent loads which took only milliseconds. I suppose this indicates that the certificate check is only done when the browser notices changes to the certificate.

Attempting to reach the same server (our locally hosted web server) via a different URL (localhost) results in a trust notice by Firefox. This is likely because the certificate we created was exclusively for a certain domain, and thus doesn't apply to localhost:4433, and wouldn't apply to any aliases that that domain operated under (most likely CNAMEs as well?).

3.4

The effective date is verified by the SSL_connect function in the client, where the line reads "err = SSL_connect (ssl);". This was determined by changing the system date to be a period when the certificate was not effective.

Whether the server certificate is signed by the authorized CA specified was also handled by the same "err = SSL_connect (ssl);" line. This was tested by entering in a server certificate that was not signed by the CA specified. Interestingly, both the effective date error and this error fail on the same line in some obscure library file for implementing SSL (probably imported by the header file "openssl/ssl.h").

Whether the certificate "belongs" to the server was tested by having the server load a private key that does not correspond to the public key it specified. The server does some checks to see if there's a mismatch between the private and public key, which we get around by commenting out all exit lines, letting the server proceed blindly. The client errors due to this mismatch again on the "err = SSL_connect (ssl);" line, however it fails due to an "alert handshake failure", on a different line than the two previous errors.

Assuming that the code does not have to be modified to include the check (because nowhere is it stated that it should have to be modified), because the question states "...machine that the client wants to talk to...", this indicates that they are referring to IPs rather than domain names. I'm a little confused as to how step 4 could possibly be referring to a "wrong domain name" check, as the code is implementing exclusively a TCP application. It appears to me that there are no attempts made in the example to verify the domain, hard-coded or otherwise, as it is in fact the server's IP (the loopback address) that is hardcoded, with DNS being skipped entirely. The client also doesn't appear to do any hacky sort of reverse DNS check (which would also give people the wrong impression on the actual ordering of the DNS (via UDP) $\Rightarrow$ HTTP (via TCP) process).

Moreover, it doesn't appear as though the existing code does any check to verify that the server it's connecting to is a valid IP for the certificate it received. In reality, this is sort of done already via the DNS system, as it's expected that only servers authorized to speak for the domain name would be returned as A records. However, we know this is not true in practice, as spoofing DNS responses is remarkably easy (a race condition ensues, but that's not hard to beat if you're on the same local network). Unless the valid server IPs were hardcoded into the certificate, I'm not sure how verification would be done. I don't think server IPs ever would be hardcoded into a certificate because a domain's IP, or set of IP's, may and should be able to change frequently without worry of having to revoke and reissue certificates.

The SSL negotiation is where the client and server decide on the cipher and secret key. Those lines in the client (and the corresponding "err = SSL_accept(ssl);" in the server) are:

```
ssl = SSL_new (ctx);        CHK_NULL(ssl);
SSL_set_fd (ssl, sd);
err = SSL_connect (ssl);
```

To prevent the server code from attempting to verify the client's certificate, we must comment out the "SSL_CTX_set_verify(ctx,SSL_VERIFY_PEER,NULL);" line, and for good measure, I replaced the following line: *client_cert = SSL_get_peer_certificate (ssl); with *client_cert = NULL;.

3.6

To digitally sign and verify example.txt, the following commands were executed:

```
openssl genrsa -des3 -out privkey.pem 2048
openssl rsa -in privkey.pem -pubout > key.pub
openssl dgst -sha256 -sign privkey.pem -out example.sha256 example.txt
openssl dgst -sha256 -verify key.pub -signature example.sha256 example.txt
/* Prints Verified OK */
```

The verification obviously failed after altering the plain-text message, as the digital signature was a signature of the original plain-text, not this modified version. Digital signatures are useful because they essentially allow authentication. So long as I have a way of verifying that a public key is owned by an individual, for instance via a CA, then I can know that any signed message that is verified with that public key was sent by that individual. This process is also tamper-safe, as indicated by this example. A modification of the plain-text message (or the signed message) will be detected by verification, so long as the individual's private key is not known.