Lab 3: Exploring Identity & Authentication

Revised: February 6, 2018

Please read through the whole lab before starting it.

Background. The goal of this lab is to familiarize students with the variety of ways systems and humans authenticate each other.

Environment. It is strongly recommended that you do this lab in a Linux machine on which you have administrator access (e.g. a virtual machines). Some aspects of this lab are more difficult (or impossible) if you don't have sudo privileges, e.g. on the CSL machines.

Task I. Exploring Certificate Authority Public Key Infrastructure (CA-PKI) with OpenSSL. The resiliency of SSL/TLS connections to impersonators resides in a collection of trusted third parties, known as certificate authorities (CAs). A site administrator engages a certificate authority (CA), such as VeriSign or Thawte, to perform the necessary verification of identity and signature operations to create a digital certificate. This certificate can then be used to bootstrap trusted communications between parties (e.g. signed and encrypted email or encrypted HTTP connections).

In this task, students will become their own certificate authority (CA), able to sign (and thus, legitimize) any entity's public key, creating a certificate. Recall from lecture that a root certificate is the self-signed public key of CA that is pre-loaded into an operating system, web browser or other software that relies on the CA-PKI. Root certificates and any certificates they verify are implicitly, and unconditionally, trusted.

A. Getting started. To begin, identify where OpenSSL keeps its default configuration. On Linux machine, it could be in `/etc/ssl/openssl.cnf` ,`/usr/lib/ssl/openssl.cnf`, on a Mac it could be `/System/Library/OpenSSL/openssl.cnf`, and on the CSL machines it might be `/etc/pki/tls/openssl.cnf`. These files should be identical. Copy one them to your home directory and open it in a text editor. Find the section labeled [CA_default]. You should see:

```
dir           = ./demoCA        # Where everything is kept
certs         = $dir/certs      # Where the issued certs are kept
crl_dir       = $dir/crl        # Where the issued crl are kept
database      = $dir/index.txt # database index file.

new_certs_dir = $dir/newcerts # default place for new certs.

serial        = $dir/serial    # The current serial number
```

Create the directories and sub-directories specified by the `openssl.cnf` file (*i.e.* `demoCA/`, `demoCA/certs`, `demoCA/crl`, and `demoCA/newcerts`). For the `index.txt file`, simply create an empty file by typing within the demoCA directory:

```
touch index.txt
```

For the `serial` file, write a single integer in the file by typing in the `demoCA` directory:

```
echo 15 > serial
```

Next, in your `openssl.cnf` file in a text editor, find the following line:

```
policy = policy_match
```

and change it to

```
policy = policy_anything
```

This small change allows your CA to create signatures for any organization in any state and in any country.

B. Becoming a Certificate Authority. To be a CA, you need to generate a root certificate and signing key. You can do this by typing:

```
openssl req -new -x509 -keyout ca.key -out ca.crt -config openssl.cnf
```

This command generates a new RSA public key pair, writes the private key to `ca.key` (encrypted under a password you provide) and writes the certificate (a self-signed public key) to `ca.crt`. The `-x509` option tells openssl to create a self-signed certificate using the X.509 standard.

You will be prompted for your "organization's" information and a passphrase. The pass phrase will be used to create a symmetric key that encrypts your private key, such that it can be securely stored on an untrusted disk. Remember this passphrase–you'll need it again!

Inspect the X.509 fields of the private key and certificate you just created. You can do so by typing:

```
openssl x509 -in ca.crt -text
```

What do you observe? What fields are there?

C. Creating a Certificate. We're now ready to create and issue certificates. For this step, switch roles and pretend to be an attacker who wants to create a valid certificate for `google.com`. Follow these steps to create that certificate:

1. Generate a key pair. To get a certificate signed by a CA, you first need to create public key pair. To do so, type:

```
openssl genrsa -des3 -out google.key 1024
```

This creates a 1024-bit RSA key pair, whose private key is protected using 3DES, and stored in the file google.key. You'll be asked for another passphrase to protect the private key on disk. To view the actual components of your key pair, use the command:

```
openssl rsa -in google.key -text -noout
```

Open the key file your just created with a text editor and note your observations.

2. Generate a Certificate Signing Request (CSR). Once you have generated your key pair, generate a Certificate Signing Request (CSR). CSRs are a standard message format that is sent from a server to a CA in order to apply for a digital certificate. It contains, among other things, the public key and the claimed identity of the applicant. A CA will use the CSR to create a valid certificate for the applicant, given the CA is convinced the identity of the applicant is valid. To create a CSR for the RSA key pair you just generated, type the following:

```
openssl req -new -key google.key -out google.csr -config openssl.cnf
```

Important: Be sure to put `google.com` as the "`Common Name (ie, YOUR name)`" of the certificate request, when prompted. This is your claimed identity. Your CSR is now in a file called `google.csr`. This is what you would send to a CA to be issued a certificate.

3. Generate the Certificate. Return to your role as the CA. You've just received a CSR from Google, and you're very excited to have such an important, first customer. You don't want to make Larry or Sergey wait, and you're *pretty* sure that CSR came from a valid Google source. No need to verify their identity, so create the certificate by typing:

```
openssl ca -in google.csr -out google.crt -cert ca.crt -keyfile ca.key
-config openssl.cnf
```

This signs the public key (and other information) you received from "Google" in their CSR using your CA's private key, creating Google's new certificate, `google.crt`. This is what you would return to the applicant for their use. *What do you observe in this certificate? Try to connect your observations to the earlier outputs. For example, where did the the serial number of the certificate come from?*

Think about what you've just done by fulfilling this CSR as a CA. *What capabilities have you just given an entity by signing their public key? What prevents anyone from impersonating Google or other well-known service in this way?* You will verify these hypotheses in the next task.

Task II. Using CA PKI to Secure and Authenticate Web Sites. In this task, you will explore how public key certificates are used by web browsers and web servers to secure and authenticate web browsing. We will use our certificate from the previous tasks, to assume the role of the attacker from Task I who wants to run a website that impersonates `google.com`. We will need to stand up a fraudulent website, intercept legitimate requests and send them to our site, and then use our certificate to perform some impersonation.

A. Intercept Requests to Google. There are multiple techniques in the real world to interpose oneself, and impersonate a website when no security mechanisms are in use. For example, we can attack the domain name system (*e.g.* using DNS cache poisoning) to cause the name '`google.com`' to map to the attacker's machine address. Alternatively, the attacker can directly interpose themselves into the path of a request (via route stealing, packet injection, *etc*). We will "fake" this.

On a Linux machine you control, open the file `/etc/hosts` with a text editor. You will need to open this file as privileged user, add the `sudo` command before your editor command (*e.g.* `sudo emacs /etc/hosts`). Add the following line to the start of your hosts file:

```
127.0.0.1   google.com
```

This overrides your system's normal DNS lookup, and will resolve all request to `google.com` to your machine. `127.0.0.1` is a special IP address that means `localhost`, or *this* machine. You can verify this is working by opening a web browser and navigating to `google.com` (Note: `www.google.com`, is a different name). Your web browser will probably return an error if no web server is running on your machine.

B. Run a Fraudulent Website using our Certificate. Now, download the `google.html` and `logo.png` files from the class website. You will use these to make your attack more convincing.

Launch a web server using the fraudulent certificate we created for `google.com` (*i.e.* `google.crt`). OpenSSL provides a simple HTTPS web server for testing purposes called `s_server`. First, create a PEM-formatted certificate file `s_server` understands by typing:

```
cat google.key google.crt > google.pem
```

Then launch the web server by typing:

```
sudo openssl s_server -accept 443 -cert google.pem -www -WWW
```

Make sure `google.html` and `logo.png` are in your present working directory. Now access your web server using the following URL: `https://google.com/google.html` (don't forget the `https`). In order to bind to SSL/TLS's default port (443) you'll need to run this as root (using sudo). You can bind to other, unrestricted ports by modifying the `-accept` switch. You will likely get an error message from your browser complaining that the certificate is not trusted because it was signed by an unknown certificate authority.

Had your `google.crt` been assigned by VeriSign, you would not have seen this error, as VeriSign's root certificate is preloaded into your browser. *Can you verify this?* Unfortunately, the CA you used is not recognized by any browsers, including Firefox. There are two ways one can get their CA root certificate into Firefox.

1. You can request Mozilla (the makers of Firefox) to include it in their release, such as these people did: https://bugzilla.mozilla.org/show_bug.cgi?id=647959. This will allow any person who uses Firefox to recognize and trust your CA. This is indeed what real CAs, such as VeriSign, do to get their root (or other) certificates included in Firefox. You can try this approach, but remember, your lab report is report is due in two weeks!

2. You can manually load your CA root certificate into your browser. For Firefox, under: `Edit > Preferences > Advanced > Certificates > View Certificates.` you will see a list of all the certificates accepted (and trusted) by Firefox. From here, we can "Import" your CA root certificate. Do this by importing `ca.crt` and select "Trust this CA to identify web sites". The CA certificate you created is now in Firefox's list of accepted certificates.

C. Test it Out. Try revisiting: `https://google.com/google.html`. *What do you observe?* Click on the navigation bar to get data about the certificate of the website. You should not see anything about an "exception" related to the certificate. (If you did, then you accidentally added an exception the first time you visited the website, and you should remove it.)

Using a different machine or browser, compare this to the real `https://www.google.com/`. *Given all that you've done, is there any way for the user or the web browser to know you're not at the real google.com?*

Lastly, since `google.com` points to `localhost`, if we use `https://localhost:443` instead, we will be connecting to the same web server. *Please do so, describe and explain your observations. What changes would you need to make to fix any problems you observe?*

Task III. Password Files. In this task, you will briefly examine how your Linux system manages and stores user passwords.

Open the file `/etc/passwd` with a text editor. Here you will see a list of all the users that have login access to your machine. You should notice a line with your username in it. `/etc/passwd` is where UNIX-based operating system used to store password hashes, but you'll notice that no hashes exist in this file. Your Linux distribution uses the shadow file: a special file managed by the operating system to store and protect password hashes.

Your shadow password exists at `/etc/shadow`. Try to open it with a text editor as an unprivileged user. *What do you observe?*

Try opening the shadow password file with root privileges–you should now be able to read the shadow password file. **BE CAREFUL!** Modifying the shadow password file can result in your inability to log in. Do not modify the shadow password file unless you know what you're doing.

Read about the shadow file by typing:

```
man 5 shadow
```

What is the hash of your password? What is the date of your last password change?

Now type:

```
man 3 crypt
```

Scroll to the Notes section and read about the algorithms used to protect your password. *What algorithm was used to protect your password? What is the value of the salt used to protect your password?*

Task IV. Password Cracking. In this task, you will try different mechanisms to crack a password file to gain access to restricted website.

A. Getting started. Download the password file (cleverly named `passwordfile.txt`) from the class website. You'll notice that this is not, in fact, a UNIX-like password file. It is actually an htpasswd file, which can be used by an Apache web server to provide password-based access control to portions of a web site. This particular .htpasswd file is from https://znjp.com/csc321, where each user has a password protected folder. For example, you would access ellie's folder by visiting https://znjp.com/csc321/ellie. Visit this site now and observe that a valid username and password is required to access this portion of the website.

Before cracking the passwords, view the contents of `passwordfile.txt`. For example, the line:

```
newton:{SHA}zfVH7Uxk5plK81z81pxCBMkiepc=
```

says the user `newton` has their password protected by the SHA1 hash function and the base64 encoded hash is `zfVH7Uxk5plK81z81pxCBMkiepc=`.

By just looking at the hashes, what can you learn about an individual user's passwords? Is there anything you can learn about the passwords of all the users from this password file?

B. Simple Dictionary Attack. Write a simple, dictionary password cracker. Your program should do the following things:

1. Open a dictionary file. Start by using the default dictionary in your Linux distribution at: `/usr/share/dict/words`. If, for some reason, you do not have a `/usr/share/dict/words` file, use the `american-english.txt` file on the class website.
2. For each word in the dictionary:
    a. Compute the SHA1 hash of the word. **Important**: Make sure you don't include the newline character in the word when hashing it.
    b. Encode the digest using base64 encoding. Be sure to encode the binary digest (*e.g.* as returned by `hashlib.sha1(word).digest()`) and not the hexadecimal representation of the digest (*e.g.* as returned by `hashlib.sha1(word).hexdigest()`.)
    c. Compare your encoded hash with your target password hashes. Be sure to include the "=" character in your target hash; it is part of the base64 encoding.
    d. If your computed hash matches any of the target hashes, you have successfully cracked a password. Display the word you used to compute the hash and quit. Otherwise, continue processing.

You can check the correctness of your code by using the Linux dictionary and the target hash `qUqP5cyxm6YcTAhz05Hph5gvu9M=`. This is a SHA1 hash of the word: `test`.

When your code is correctly computing hashes, use the password hashes from `passwordfile.txt` in your program and attempt to crack them. *If your program completes (*i.e.* hashes all the words in the dictionary) without finding a match, what can you assume about the passwords? Which passwords were you able to crack?*

C. Common Password Dictionary Attack. By now, you may have cracked some passwords, but not all. This is because the dictionary you used contains only English words, and no common passwords or variants thereof (*e.g.* replacing i's with 1's, e's with 3's, *etc.*). Download the `passwords-dict.txt` file from the class website. This file contains 1.7 million commonly used passwords and some of their variants. (As a security exercise, see if any passwords you commonly use are in this file!)

Modify your program to use your new password dictionary and try cracking the remaining passwords. You should be able to crack a few more. *Which additional passwords did you crack?*[1]

Were you able to crack all the passwords? Why or why not? If not, what characteristics do you think the un-cracked password has? For those passwords you were unable to crack, what would be required to crack them and how do you think it would take?

Using each of the cracked passwords, go to the user's respective web page. *What is the secret word for each user?*

Submission. Write a brief report describing what you did and what you observed. Include any code that you wrote, as well as answers to any questions (these are in italics), within the report itself. Please include any explanations of the surprising or interesting observations you made.

Write at a level that demonstrates your technical understanding, and do not shorthand ideas under the assumption that the reader already "knows what you mean." Think of writing as if the audience was a smart colleague who may not have taken this class.

Describe what you did in sufficient detail that it can be reproduced. Please do not use screenshots of the VM to show the commands and code you used, but rather paste (or carefully re-type) these into your report from your terminal. Submit your completed write up to PolyLearn in PDF format.

[1] If you didn't crack any more passwords, make sure you got the complete passwords-dict.txt file. You can verify you did by typing the command: `wc passwords-dict.txt` and see the number of lines if around 1.7 million.