Justin Gallagher

CSC 482

Crypto PKI Lab

Public key cryptographic algorithms are highly secure until we can build a large quantum computer or prove that P=NP, public key infrastructure has a variety of weaknesses that have been exploited by attackers. In this lab, we will become a certificate authority (CA), create certificates, and explore MITM attacks, both with and without violating CA security.

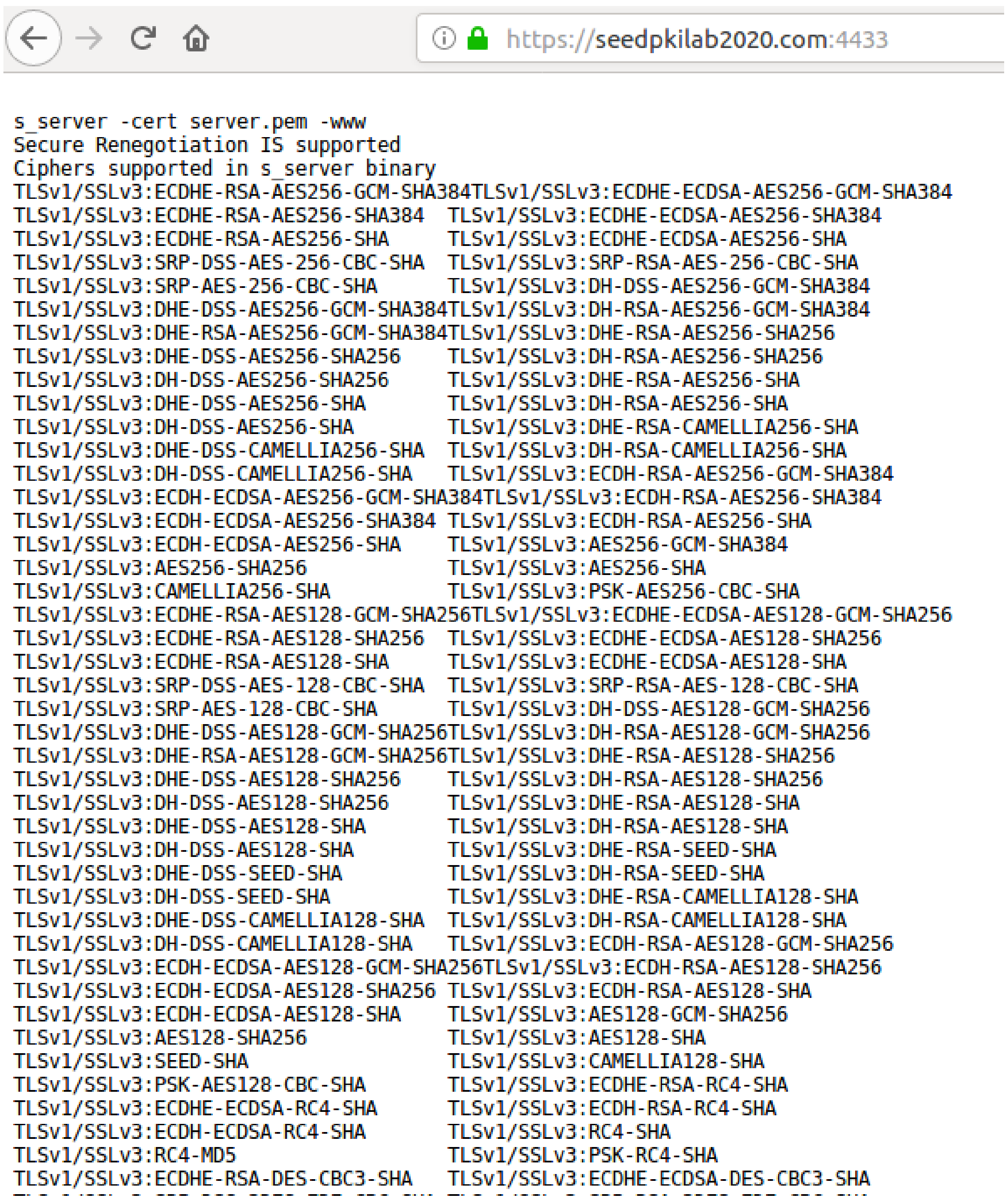
Task 2: Creating a Certificate for SEEDPKILab2020.com

In this task, we sign digital certificates for a fake customer company called ‘SEEDPKILab2020.com’. To get a digital certificate from a CA, we must follow three steps. We assume that the first step has been completed, which involved unziping the ‘pki’ file downloaded from the canvas page and running the set-up shell script to become a CA. Now, the first step is to generate the public/private key pair. There is a command that I used from the lab that creates an RSA key pair. Upon using the command, the keys are stored in the file ‘server.key’. Once the company has the key file, the next step is to generate a Certificate Signing Request (CSR), which basically includes the company’s public key. The CSR will be sent to the CA, who will generate a certificate for the key (usually after ensuring that identity information in the CSR matches with the server’s identity). The last step involves generating certificates. The CSR file needs to have the CA’s signature to form a certificate. In this lab, we will use our CA to generate certificates. The three commands used in this task are listed below:

```
openssl genrsa -aes128 -out server.key 1024
openssl req -new -key server.key -out server.csr -config openssl.cnf
openssl ca -in server.csr -out server.crt -cert ca.crt -keyfile ca.key -config openssl.cnf
```

Task 3: Deploying Certificate in a HTTPS Server

In this task, we set up and deploy a certificate for a HTTPS server. To do this we first need to set up our system to recognize the name of our website, ‘SEEDPKILab2020.com’. This can be done by editing the /etc/hosts file using vim to include the entry for our hostname. Once this is done, we need to configure the web server. To do this, I used a series of commands that combine the secret key and certificate into one file called ‘server.pem’ and then use this file to launch a web server. Once this was done, the server is now running. We can now attempt to access the website using the following URL: ‘https://seedpkilab2020.com:4433/’. There is a slight problem though. While attempting to access the website, there is an error. This error arises because Firefox, our browser, does not accept our CA certificate. To allow Firefox to accept our browser, we can import the ‘ca.crt’ file created in Task2 into the allowed certificates list that Firefox holds. After doing this, I load up the website and I am met with this screen:



https://seedpkilab2020.com:4433

```
s_server -cert server.pem -www
Secure Renegotiation IS supported
Ciphers supported in s_server binary
TLSv1/SSLv3:ECDHE-RSA-AES256-GCM-SHA384TLSv1/SSLv3:ECDHE-ECDSA-AES256-GCM-SHA384
TLSv1/SSLv3:ECDHE-RSA-AES256-SHA384  TLSv1/SSLv3:ECDHE-ECDSA-AES256-SHA384
TLSv1/SSLv3:ECDHE-RSA-AES256-SHA     TLSv1/SSLv3:ECDHE-ECDSA-AES256-SHA
TLSv1/SSLv3:SRP-DSS-AES-256-CBC-SHA  TLSv1/SSLv3:SRP-RSA-AES-256-CBC-SHA
TLSv1/SSLv3:SRP-AES-256-CBC-SHA      TLSv1/SSLv3:DH-DSS-AES256-GCM-SHA384
TLSv1/SSLv3:DHE-DSS-AES256-GCM-SHA384TLSv1/SSLv3:DH-RSA-AES256-GCM-SHA384
TLSv1/SSLv3:DHE-RSA-AES256-GCM-SHA384TLSv1/SSLv3:DHE-RSA-AES256-SHA256
TLSv1/SSLv3:DHE-DSS-AES256-SHA256    TLSv1/SSLv3:DH-RSA-AES256-SHA256
TLSv1/SSLv3:DH-DSS-AES256-SHA256     TLSv1/SSLv3:DHE-RSA-AES256-SHA
TLSv1/SSLv3:DHE-DSS-AES256-SHA       TLSv1/SSLv3:DH-RSA-AES256-SHA
TLSv1/SSLv3:DH-DSS-AES256-SHA        TLSv1/SSLv3:DHE-RSA-CAMELLIA256-SHA
TLSv1/SSLv3:DHE-DSS-CAMELLIA256-SHA  TLSv1/SSLv3:DH-RSA-CAMELLIA256-SHA
TLSv1/SSLv3:DH-DSS-CAMELLIA256-SHA   TLSv1/SSLv3:ECDH-RSA-AES256-GCM-SHA384
TLSv1/SSLv3:ECDH-ECDSA-AES256-GCM-SHA384TLSv1/SSLv3:ECDH-RSA-AES256-SHA384
TLSv1/SSLv3:ECDH-ECDSA-AES256-SHA384 TLSv1/SSLv3:ECDH-RSA-AES256-SHA
TLSv1/SSLv3:ECDH-ECDSA-AES256-SHA    TLSv1/SSLv3:AES256-GCM-SHA384
TLSv1/SSLv3:AES256-SHA256            TLSv1/SSLv3:AES256-SHA
TLSv1/SSLv3:CAMELLIA256-SHA          TLSv1/SSLv3:PSK-AES256-CBC-SHA
TLSv1/SSLv3:ECDHE-RSA-AES128-GCM-SHA256TLSv1/SSLv3:ECDHE-ECDSA-AES128-GCM-SHA256
TLSv1/SSLv3:ECDHE-RSA-AES128-SHA256  TLSv1/SSLv3:ECDHE-ECDSA-AES128-SHA256
TLSv1/SSLv3:ECDHE-RSA-AES128-SHA     TLSv1/SSLv3:ECDHE-ECDSA-AES128-SHA
TLSv1/SSLv3:SRP-DSS-AES-128-CBC-SHA  TLSv1/SSLv3:SRP-RSA-AES-128-CBC-SHA
TLSv1/SSLv3:SRP-AES-128-CBC-SHA      TLSv1/SSLv3:DH-DSS-AES128-GCM-SHA256
TLSv1/SSLv3:DHE-DSS-AES128-GCM-SHA256TLSv1/SSLv3:DH-RSA-AES128-GCM-SHA256
TLSv1/SSLv3:DHE-RSA-AES128-GCM-SHA256TLSv1/SSLv3:DHE-RSA-AES128-SHA256
TLSv1/SSLv3:DHE-DSS-AES128-SHA256    TLSv1/SSLv3:DH-RSA-AES128-SHA256
TLSv1/SSLv3:DH-DSS-AES128-SHA256     TLSv1/SSLv3:DHE-RSA-AES128-SHA
TLSv1/SSLv3:DHE-DSS-AES128-SHA       TLSv1/SSLv3:DH-RSA-AES128-SHA
TLSv1/SSLv3:DH-DSS-AES128-SHA        TLSv1/SSLv3:DHE-RSA-SEED-SHA
TLSv1/SSLv3:DHE-DSS-SEED-SHA         TLSv1/SSLv3:DH-RSA-SEED-SHA
TLSv1/SSLv3:DH-DSS-SEED-SHA          TLSv1/SSLv3:DHE-RSA-CAMELLIA128-SHA
TLSv1/SSLv3:DHE-DSS-CAMELLIA128-SHA  TLSv1/SSLv3:DH-RSA-CAMELLIA128-SHA
TLSv1/SSLv3:DH-DSS-CAMELLIA128-SHA   TLSv1/SSLv3:ECDH-RSA-AES128-GCM-SHA256
TLSv1/SSLv3:ECDH-ECDSA-AES128-GCM-SHA256TLSv1/SSLv3:ECDH-RSA-AES128-SHA256
TLSv1/SSLv3:ECDH-ECDSA-AES128-SHA256 TLSv1/SSLv3:ECDH-RSA-AES128-SHA
TLSv1/SSLv3:ECDH-ECDSA-AES128-SHA    TLSv1/SSLv3:AES128-GCM-SHA256
TLSv1/SSLv3:AES128-SHA256            TLSv1/SSLv3:AES128-SHA
TLSv1/SSLv3:SEED-SHA                 TLSv1/SSLv3:CAMELLIA128-SHA
TLSv1/SSLv3:PSK-AES128-CBC-SHA       TLSv1/SSLv3:ECDHE-RSA-RC4-SHA
TLSv1/SSLv3:ECDHE-ECDSA-RC4-SHA      TLSv1/SSLv3:ECDH-RSA-RC4-SHA
TLSv1/SSLv3:ECDH-ECDSA-RC4-SHA       TLSv1/SSLv3:RC4-SHA
TLSv1/SSLv3:RC4-MD5                  TLSv1/SSLv3:PSK-RC4-SHA
TLSv1/SSLv3:ECDHE-RSA-DES-CBC3-SHA   TLSv1/SSLv3:ECDHE-ECDSA-DES-CBC3-SHA
```

Once we are at this point, the last part of this task has us testing the site. The first test is to modify a single bit from the 'server.pem' file and then restart the server and note any changes to the site. Upon doing these things, I noticed that after restarting the server and refreshing the webpage, Firefox would give this error:

An error occurred during a connection to seedpkilab2020.com:4433. You have received an invalid certificate. Please contact the server administrator or email correspondent and give them the following information: Your certificate contains the same serial number as another certificate issued by the certificate authority. Please get a new certificate containing a unique serial number. Error code: SEC_ERROR_REUSED_ISSUER_AND_SERIAL

The terminal also throws this error:

```
3070600896:error:14094412:SSL routines:ssl3_read_bytes:sslv3 alert bad certificate:s3_pkt.c:1487:SSL alert number 42

3070600896:error:140780E5:SSL routines:ssl23_read:ssl handshake failure:s23_lib.c:137:
```

The second test is to use 'https://localhost:4433' instead to connect and determine whether we are connecting to the same webserver. Upon using the new address, we see that we are connecting to the same webserver because when doing so, the terminal outputs 'ACCEPT' for every connection made and will output when we connect using localhost. Commands used for this task are listed below:

```
sudo vim /etc/hosts
cp server.key server.pem
cat server.crt >> server.pem
openssl s_server –cert server.pem -www
```

Task 4: Deploying Certificate in an Apache-Based HTTPS Website

In this task, we set up an Apache web server and enable HTTPS for the SEEDPKILab2020.com website. To do this, I start by creating a file called seedpkilab2020_com.conf and giving it the contents offered by the lab. I then created a directory called /var/www/seedpkilab2020_com and made sure its permissions matched the /var/www/html permissions. I then made another file called seedpkilab2020_ssl.conf and filled it with the contents from the lab. I modified the _com and _ssl files so that they would point to the right directories, as well as I added a new directory called /etc/apache2/ssl to store the server.key file and the server/crt file. I then used a command to get rid of the password that pertained to the key. After all of this was finsihed, I made a quick html file to test the webpage out. I then restarted the server and used a a2ensite command to enable to web site file. Upon accessing the websites URL, I was met with the expected html page proving that this method works. All files can be found in their appropriate folders on the VM, and commands used are listed below:

```
sudo vim … (this command was used a lot)
openssl rsa -in example.key -out example-nopw.key
mv example-nopw.key example.key
sudo a2ensite example_ssl.conf
sudo apachectl configtest
sudo apachectl restart
```

Task 5: Deploying Certificate in an Apache-Based HTTPS Website

In this task, we will simulate a MITM attack to understand how PKI can defeat such an attack. To do this, we will use our apache server from task 4 to set up and impersonate another website. I first began by creating keys and certificates for the impersonated site, in this case we will impersonate the site we created in the last task, the SEEDPKILab2020.com site. I then create the configuration file for the site and made an identical homepage and stored it in DocumentRoot. Then, I enabled the site and restarted the server. For the victim to land on our webpage, we must modify the victim’s machine so that its /etc/hosts file points to the IP address of the attacker’s web server. Trying to load up the website for SEEDPKILab2020.com results in an error saying that the connection is not secure. This shows that the MITM attack is successfully stopped by the web browser because although the attackers certificate passes as valid, the browser needs to know the domain name of the target website matches that of the victim's request. Commands used in this task are the same used in the previous tasks.