Lab 3 - Playing with PKI

Team Members:

1. Adam Robertson, abr5598@psu.edu, 938152440

Drills

There are five tasks for you to complete. Please give a brief summary of what you did – feel free to include any thoughts / concerns / problems / etc. you encountered during the tasks. Also, include your answers to the questions asked in each task. Save your report as a PDF and submit it to Canvas before the deadline.

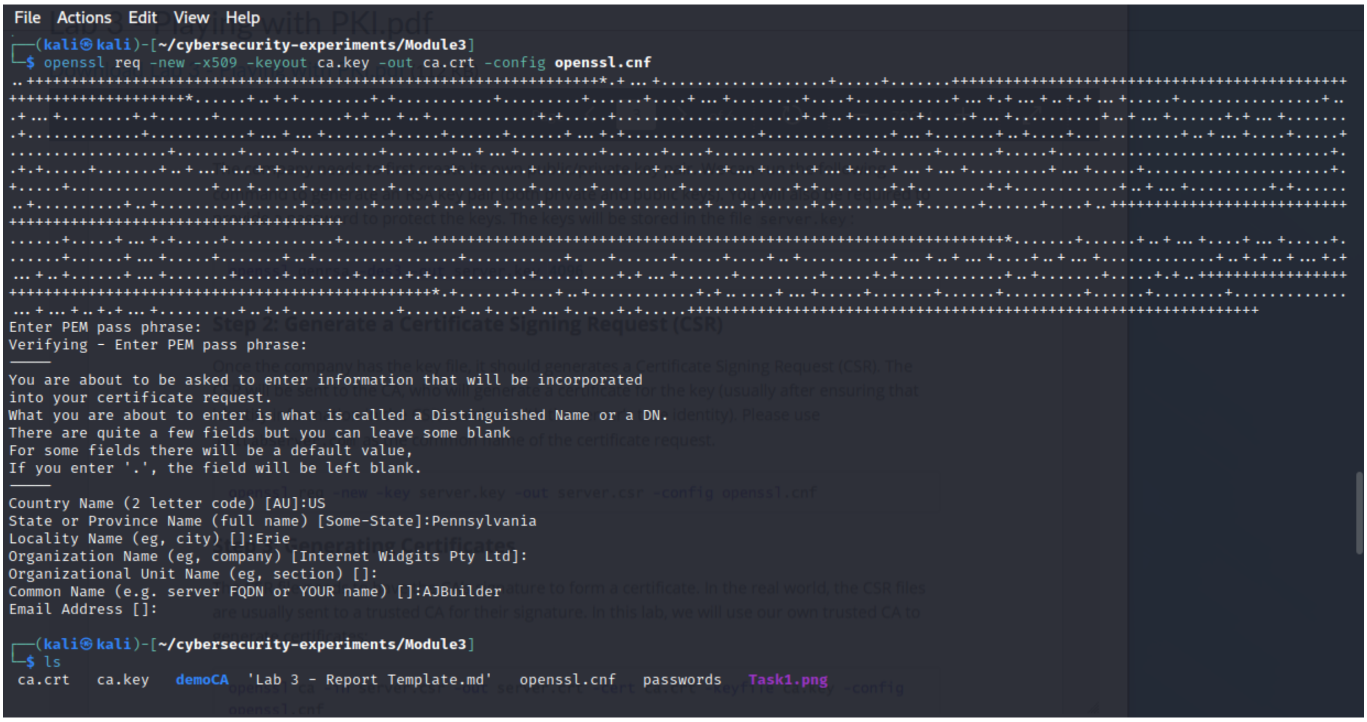
Task 1

Task 1: Summary

In task 1, we are becoming a Certificate Authority by generating our own root certificate.

Task 1: Question Answers

> 1. Include the screenshot of your operations, such as commands and output.



```
┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3]
└─$ openssl req -new -x509 -keyout ca.key -out ca.crt -config openssl.cnf
..+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*.+...+....................+.....+.......+++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++*......+..+.+.........+.+.............+..........+.......+.....+...+....+...+.........+.....+............+...+.+...+..+.+...+......+.................+..
.+...+.........+.+.......+................+.+...+..+..............+.+.....+...........................+.+..+..........+.......+....+.+...+..+...+.......+.+...+.......
.+...............+............+...+...+........+......+.......+.......+...+.+.................+..................+...+........+..+....+...............+..+...+....+.....+
...................+........+......+..........+........+..+...+..........+.......+.....+....+.....+......................+.......+.......+.....+.................................+.
.+.+......+........+..+...+...+...+.+.............+.........+..........+........+..+.+...+.+...+...+...+...........+...+......+.........................+.
+......+.................+...+......+..........+.................+.......+.........+...........+.+...............+.+.............+..+...+.........+.+......
..+...........+..+..........+.......+........+.......+..+...+.....+.......+...............+...+..+.........+.........+.....+..++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++
......+.....+...+.+.....+.............+........+..+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*.......+.......+..+...+.....+...+.......+.
......+......+...+.....+........+..+.................+............+....+......+...+.....+....+..+...........+...+..+...+.....+..+...+.............+..+.+..+...+.+
...+..+......+...+.......+.........+..........+......+.+.............+..+.........+.....+.+...+.......+..........+......+.+..+++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++*.+.......+....+..+.............+......+.+.......+...+.....+.......+.......+..........+.......+.........+.............
...+...+..+.+...+.........+..+.+.............+......+..+....+...+.....+.+.....++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Pennsylvania
Locality Name (eg, city) []:Erie
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:AJBuilder
Email Address []:

┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3]
└─$ ls
 ca.crt   ca.key   demoCA  'Lab 3 - Report Template.md'   openssl.cnf   passwords   Task1.png
```

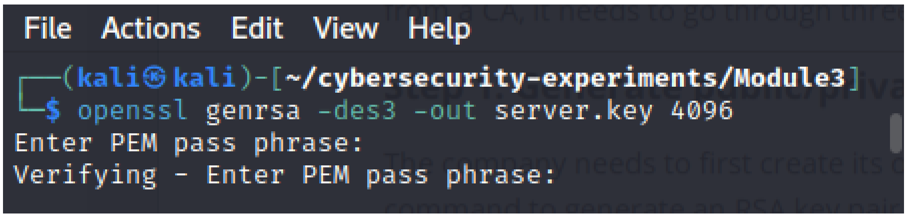
Task 2

Task 2: Summary

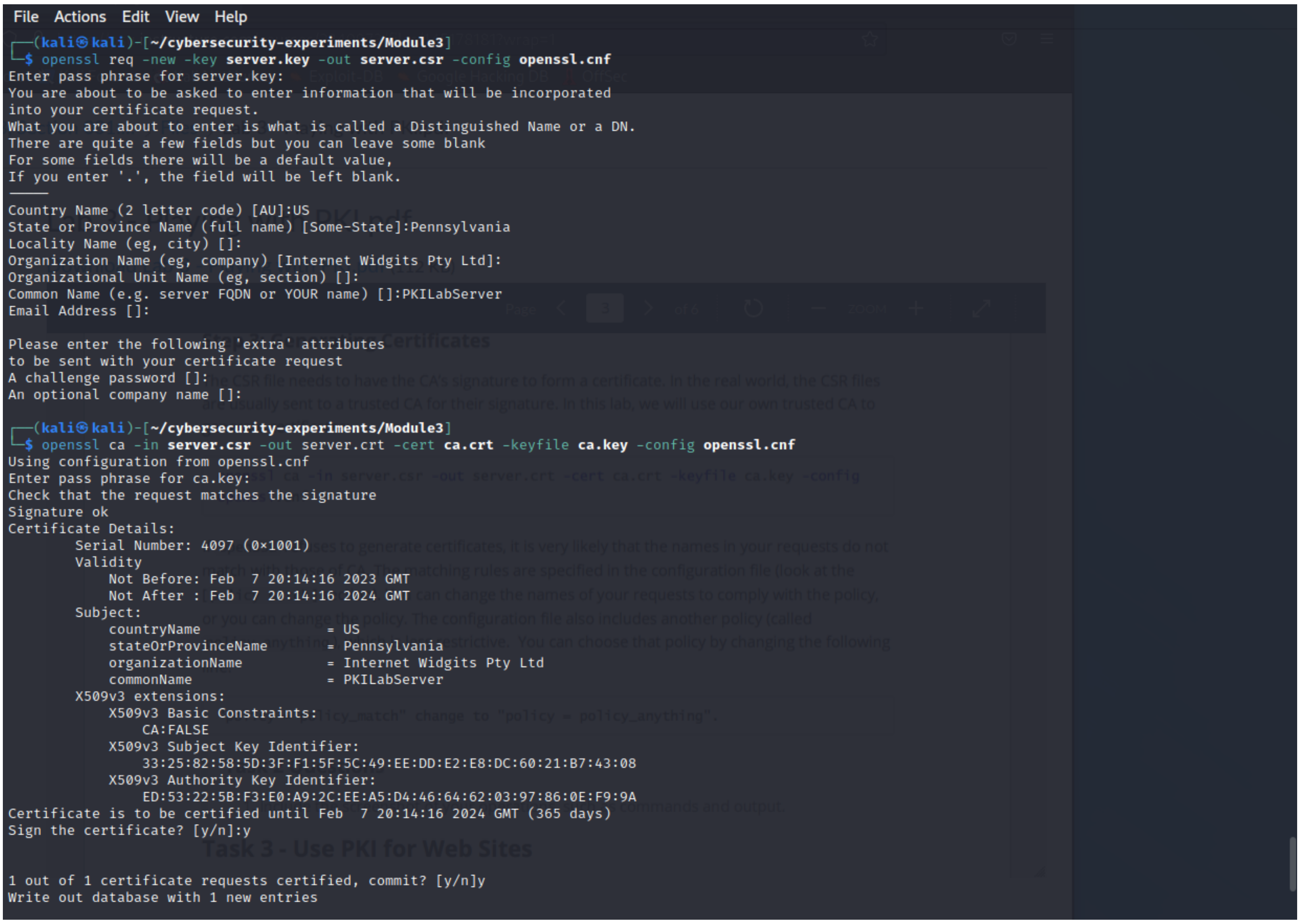
In task 2, we play the part of both the CA and a customer wanting to get a certificate signed by the CA. The customer generates a certificate signing request using their information and key. The CA then signs that certificate that the customer can then use for their server.

Task 2: Question Answers

> 1. Include the screenshot of your operations, such as commands and output.



```
┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3]
└─$ openssl genrsa -des3 -out server.key 4096
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
```

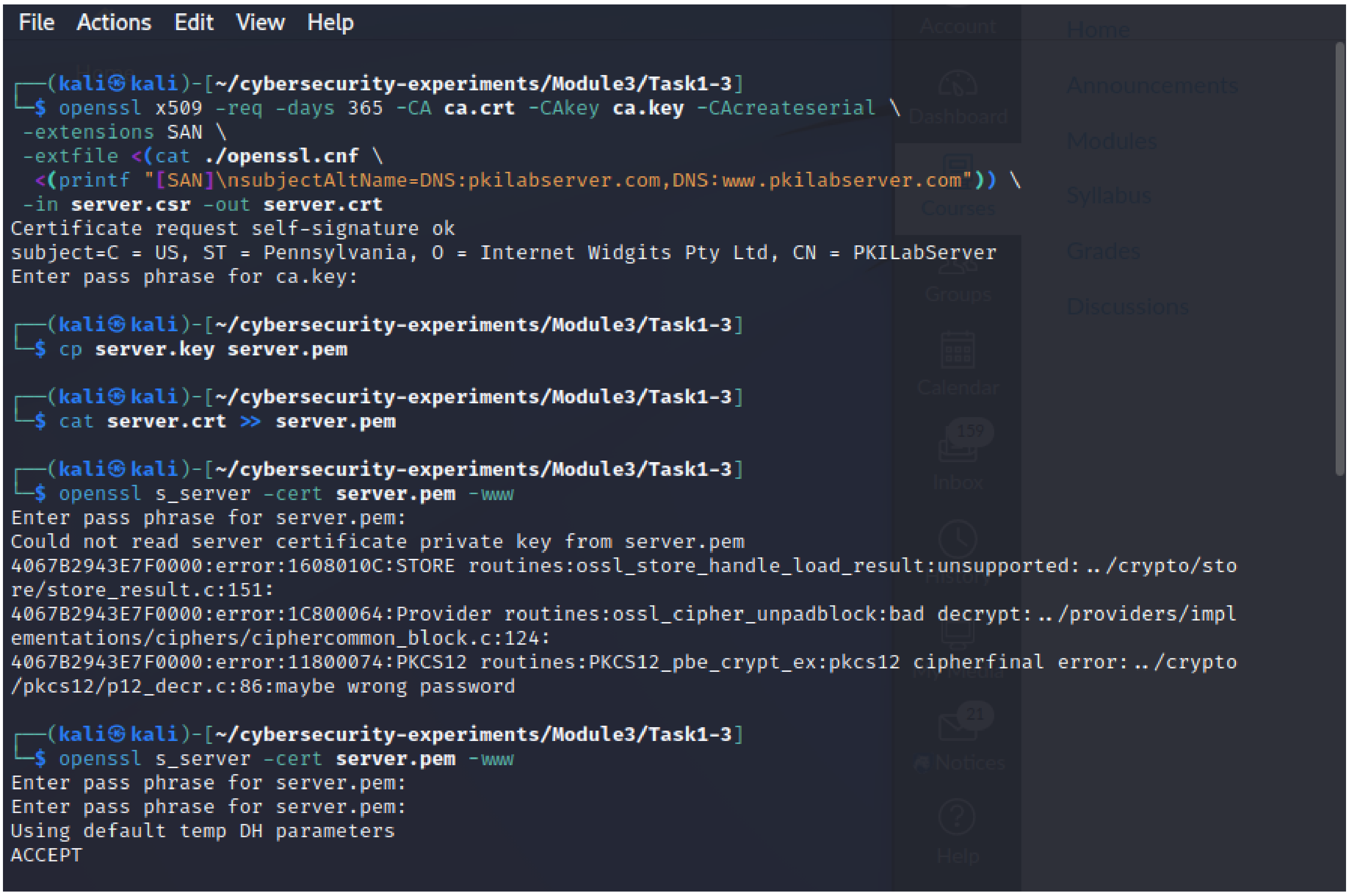


```
┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3]
└─$ openssl req -new -key server.key -out server.csr -config openssl.cnf
Enter pass phrase for server.key:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
─────
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Pennsylvania
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:PKILabServer
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:

┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3]
└─$ openssl ca -in server.csr -out server.crt -cert ca.crt -keyfile ca.key -config openssl.cnf
Using configuration from openssl.cnf
Enter pass phrase for ca.key:
Check that the request matches the signature
Signature ok
Certificate Details:
        Serial Number: 4097 (0×1001)
        Validity
            Not Before: Feb  7 20:14:16 2023 GMT
            Not After : Feb  7 20:14:16 2024 GMT
        Subject:
            countryName               = US
            stateOrProvinceName       = Pennsylvania
            organizationName          = Internet Widgits Pty Ltd
            commonName                = PKILabServer
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Subject Key Identifier:
                33:25:82:58:5D:3F:F1:5F:5C:49:EE:DD:E2:E8:DC:60:21:B7:43:08
            X509v3 Authority Key Identifier:
                ED:53:22:5B:F3:E0:A9:2C:EE:A5:D4:46:64:62:03:97:86:0E:F9:9A
Certificate is to be certified until Feb  7 20:14:16 2024 GMT (365 days)
Sign the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
```



```
┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task1-3]
└─$ openssl x509 -req -days 365 -CA ca.crt -CAkey ca.key -CAcreateserial \
 -extensions SAN \
 -extfile <(cat ./openssl.cnf \
  <(printf "[SAN]\nsubjectAltName=DNS:pkilabserver.com,DNS:www.pkilabserver.com")) \
 -in server.csr -out server.crt
Certificate request self-signature ok
subject=C = US, ST = Pennsylvania, O = Internet Widgits Pty Ltd, CN = PKILabServer
Enter pass phrase for ca.key:

┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task1-3]
└─$ cp server.key server.pem

┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task1-3]
└─$ cat server.crt >> server.pem

┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task1-3]
└─$ openssl s_server -cert server.pem -www
Enter pass phrase for server.pem:
Could not read server certificate private key from server.pem
4067B2943E7F0000:error:1608010C:STORE routines:ossl_store_handle_load_result:unsupported:../crypto/sto
re/store_result.c:151:
4067B2943E7F0000:error:1C800064:Provider routines:ossl_cipher_unpadblock:bad decrypt:../providers/impl
ementations/ciphers/ciphercommon_block.c:124:
4067B2943E7F0000:error:11800074:PKCS12 routines:PKCS12_pbe_crypt_ex:pkcs12 cipherfinal error:../crypto
/pkcs12/p12_decr.c:86:maybe wrong password

┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task1-3]
└─$ openssl s_server -cert server.pem -www
Enter pass phrase for server.pem:
Enter pass phrase for server.pem:
Using default temp DH parameters
ACCEPT
```

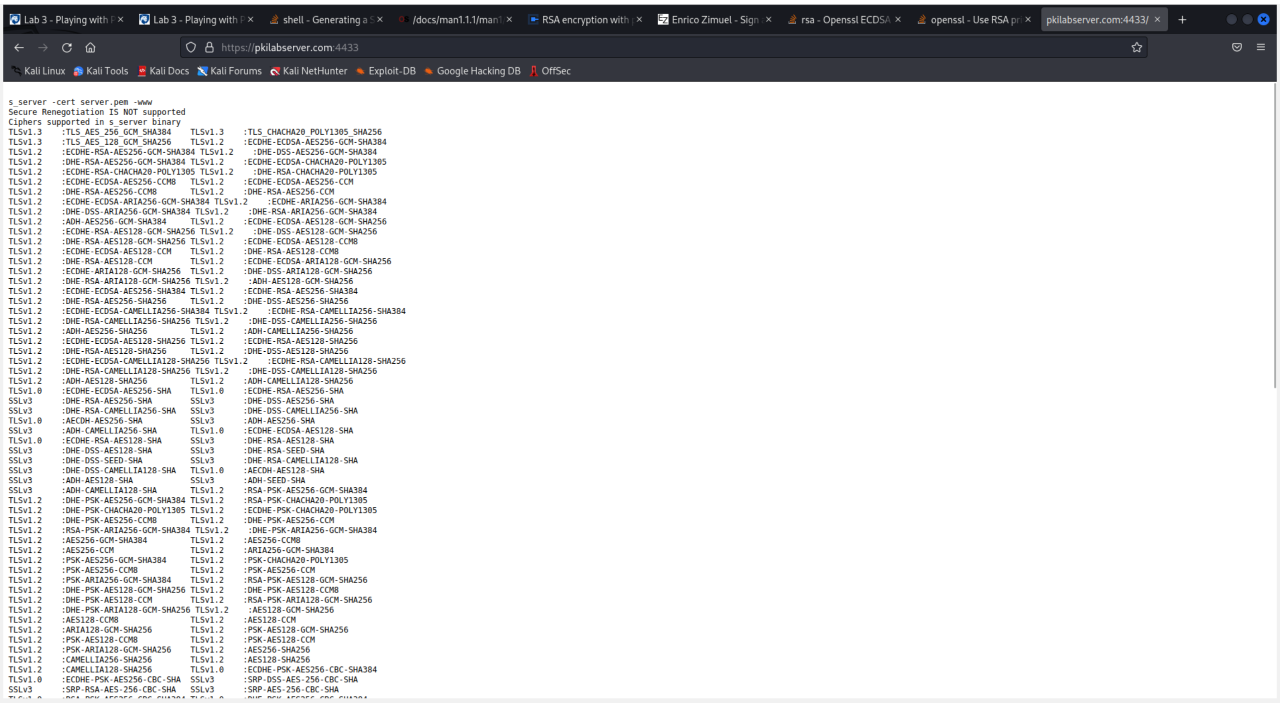
Task 3

Task 3: Summary

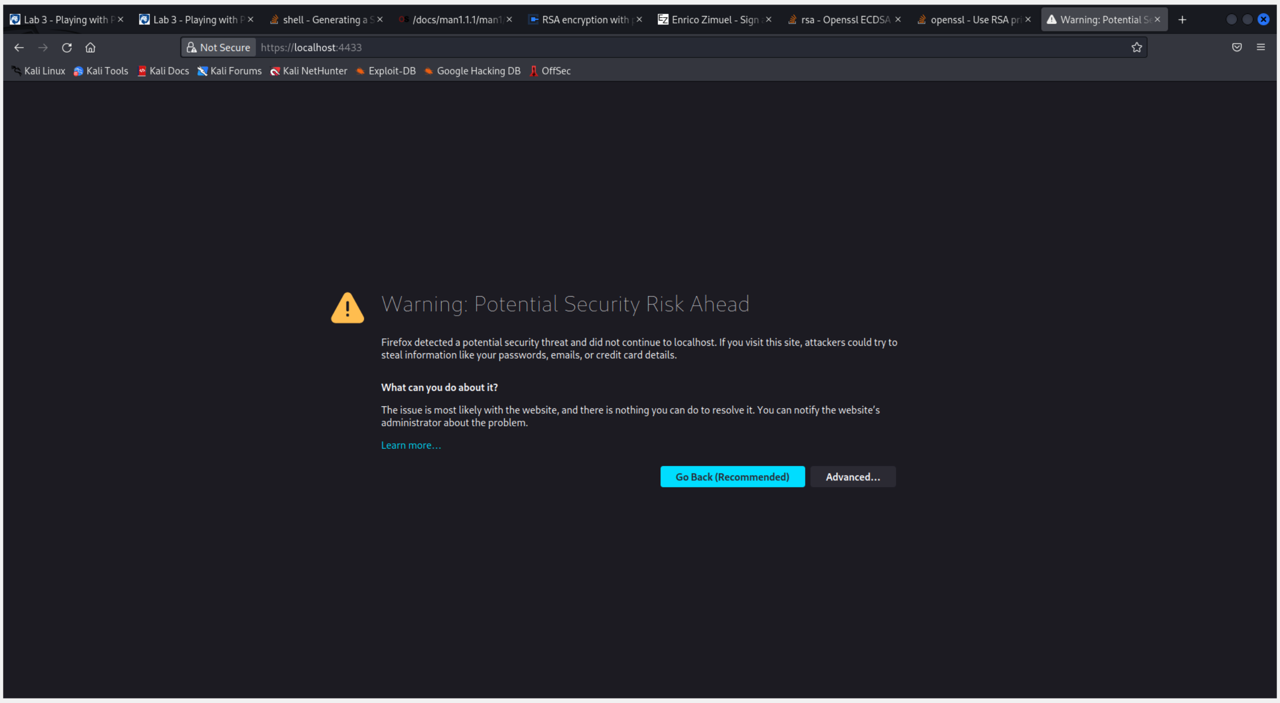
In Task 3, we load the root certificate generated in task 1 into our web browser. We then try to connect to a web server that has a certificate signed by the root certifiacte we loaded so we can verify their identity. (We are also hosting the web browser.)

Task 3: Question Answers

> *The screenshot of your operations can be included for better demo.*



```
s_server -cert server.pem -www
Secure Renegotiation IS NOT supported
Ciphers supported in s_server binary
TLSv1.3    :TLS_AES_256_GCM_SHA384    TLSv1.3    :TLS_CHACHA20_POLY1305_SHA256
TLSv1.3    :TLS_AES_128_GCM_SHA256    TLSv1.2    :ECDHE-ECDSA-AES256-GCM-SHA384
TLSv1.2    :ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2    :DHE-DSS-AES256-GCM-SHA384
TLSv1.2    :DHE-RSA-AES256-GCM-SHA384 TLSv1.2    :ECDHE-ECDSA-CHACHA20-POLY1305
TLSv1.2    :ECDHE-RSA-CHACHA20-POLY1305 TLSv1.2    :DHE-RSA-CHACHA20-POLY1305
TLSv1.2    :ECDHE-ECDSA-AES256-CCM8   TLSv1.2    :ECDHE-ECDSA-AES256-CCM
TLSv1.2    :DHE-RSA-AES256-CCM8       TLSv1.2    :DHE-RSA-AES256-CCM
TLSv1.2    :ECDHE-ECDSA-ARIA256-GCM-SHA384 TLSv1.2    :ECDHE-ARIA256-GCM-SHA384
TLSv1.2    :DHE-DSS-ARIA256-GCM-SHA384 TLSv1.2    :DHE-RSA-ARIA256-GCM-SHA384
TLSv1.2    :ADH-AES256-GCM-SHA384     TLSv1.2    :ECDHE-ECDSA-AES128-GCM-SHA256
TLSv1.2    :ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2    :DHE-DSS-AES128-GCM-SHA256
TLSv1.2    :DHE-RSA-AES128-GCM-SHA256 TLSv1.2    :ECDHE-ECDSA-AES128-CCM8
TLSv1.2    :ECDHE-ECDSA-AES128-CCM    TLSv1.2    :DHE-RSA-AES128-CCM8
TLSv1.2    :DHE-RSA-AES128-CCM        TLSv1.2    :ECDHE-ECDSA-ARIA128-GCM-SHA256
TLSv1.2    :ECDHE-ARIA128-GCM-SHA256  TLSv1.2    :DHE-DSS-ARIA128-GCM-SHA256
TLSv1.2    :DHE-RSA-ARIA128-GCM-SHA256 TLSv1.2    :ADH-AES128-GCM-SHA256
TLSv1.2    :ECDHE-ECDSA-AES256-SHA384 TLSv1.2    :ECDHE-RSA-AES256-SHA384
TLSv1.2    :DHE-RSA-AES256-SHA256     TLSv1.2    :DHE-DSS-AES256-SHA256
TLSv1.2    :ECDHE-ECDSA-CAMELLIA256-SHA384 TLSv1.2    :ECDHE-RSA-CAMELLIA256-SHA384
TLSv1.2    :DHE-RSA-CAMELLIA256-SHA256 TLSv1.2    :DHE-DSS-CAMELLIA256-SHA256
TLSv1.2    :ADH-AES256-SHA256         TLSv1.2    :ADH-CAMELLIA256-SHA256
TLSv1.2    :ECDHE-ECDSA-AES128-SHA256 TLSv1.2    :ECDHE-RSA-AES128-SHA256
TLSv1.2    :DHE-RSA-AES128-SHA256     TLSv1.2    :DHE-DSS-AES128-SHA256
TLSv1.2    :ECDHE-ECDSA-CAMELLIA128-SHA256 TLSv1.2    :ECDHE-RSA-CAMELLIA128-SHA256
TLSv1.2    :DHE-RSA-CAMELLIA128-SHA256 TLSv1.2    :DHE-DSS-CAMELLIA128-SHA256
TLSv1.2    :ADH-AES128-SHA256         TLSv1.2    :ADH-CAMELLIA128-SHA256
TLSv1.0    :ECDHE-ECDSA-AES256-SHA    TLSv1.0    :ECDHE-RSA-AES256-SHA
SSLv3      :DHE-RSA-AES256-SHA        SSLv3      :DHE-DSS-AES256-SHA
SSLv3      :DHE-RSA-CAMELLIA256-SHA   SSLv3      :DHE-DSS-CAMELLIA256-SHA
TLSv1.0    :AECDH-AES256-SHA          SSLv3      :ADH-AES256-SHA
SSLv3      :ADH-CAMELLIA256-SHA       TLSv1.0    :ECDHE-ECDSA-AES128-SHA
TLSv1.0    :ECDHE-RSA-AES128-SHA      SSLv3      :DHE-RSA-AES128-SHA
SSLv3      :DHE-DSS-AES128-SHA        SSLv3      :DHE-RSA-SEED-SHA
SSLv3      :DHE-DSS-SEED-SHA          SSLv3      :DHE-RSA-CAMELLIA128-SHA
SSLv3      :DHE-DSS-CAMELLIA128-SHA   TLSv1.0    :AECDH-AES128-SHA
SSLv3      :ADH-AES128-SHA            SSLv3      :ADH-SEED-SHA
SSLv3      :ADH-CAMELLIA128-SHA       TLSv1.2    :RSA-PSK-AES256-GCM-SHA384
TLSv1.2    :DHE-PSK-AES256-GCM-SHA384 TLSv1.2    :RSA-PSK-CHACHA20-POLY1305
TLSv1.2    :DHE-PSK-CHACHA20-POLY1305 TLSv1.2    :ECDHE-PSK-CHACHA20-POLY1305
TLSv1.2    :DHE-PSK-AES256-CCM8       TLSv1.2    :DHE-PSK-AES256-CCM
TLSv1.2    :RSA-PSK-ARIA256-GCM-SHA384 TLSv1.2    :DHE-PSK-ARIA256-GCM-SHA384
TLSv1.2    :AES256-GCM-SHA384         TLSv1.2    :AES256-CCM8
TLSv1.2    :AES256-CCM                TLSv1.2    :ARIA256-GCM-SHA384
TLSv1.2    :PSK-AES256-GCM-SHA384     TLSv1.2    :PSK-CHACHA20-POLY1305
TLSv1.2    :PSK-AES256-CCM8           TLSv1.2    :PSK-AES256-CCM
TLSv1.2    :PSK-ARIA256-GCM-SHA384    TLSv1.2    :RSA-PSK-AES128-GCM-SHA256
TLSv1.2    :DHE-PSK-AES128-GCM-SHA256 TLSv1.2    :DHE-PSK-AES128-CCM8
TLSv1.2    :DHE-PSK-AES128-CCM        TLSv1.2    :RSA-PSK-ARIA128-GCM-SHA256
TLSv1.2    :DHE-PSK-ARIA128-GCM-SHA256 TLSv1.2    :AES128-GCM-SHA256
TLSv1.2    :AES128-CCM8               TLSv1.2    :AES128-CCM
TLSv1.2    :ARIA128-GCM-SHA256        TLSv1.2    :PSK-AES128-GCM-SHA256
TLSv1.2    :PSK-AES128-CCM8           TLSv1.2    :PSK-AES128-CCM
TLSv1.2    :PSK-ARIA128-GCM-SHA256    TLSv1.2    :AES256-SHA256
TLSv1.2    :CAMELLIA256-SHA256        TLSv1.2    :AES128-SHA256
TLSv1.2    :CAMELLIA128-SHA256        TLSv1.0    :ECDHE-PSK-AES256-CBC-SHA384
TLSv1.0    :ECDHE-PSK-AES256-CBC-SHA  SSLv3      :SRP-DSS-AES-256-CBC-SHA
SSLv3      :SRP-RSA-AES-256-CBC-SHA   SSLv3      :SRP-AES-256-CBC-SHA
```



Warning: Potential Security Risk Ahead

Firefox detected a potential security threat and did not continue to localhost. If you visit this site, attackers could try to steal information like your passwords, emails, or credit card details.

What can you do about it?

The issue is most likely with the website, and there is nothing you can do to resolve it. You can notify the website's administrator about the problem.

Learn more…

1. (a) Please describe and explain your observations

The web browser did not trust the certificate provided by pkilabserver.com until we loaded the Certificate Authorities certificate.

To my understanding, since the server's certificate is signed by the CA's private key, the CA certificate we loaded into the web browser was the public key necessary to decrypt the CA signature on the servers signature. However, I don't understand why the servers private key was necesarry to generate the servers certificate in Task 2. The client never gets the servers public key so they wouldn't be able to verify the servers signature anyway.

2. (b) What do you observe?

The webserver says it "cannot read the certificate private key". Which confuses me on what the purpose of the server private key is for. Throughout the lab when refering to the server key, it is never

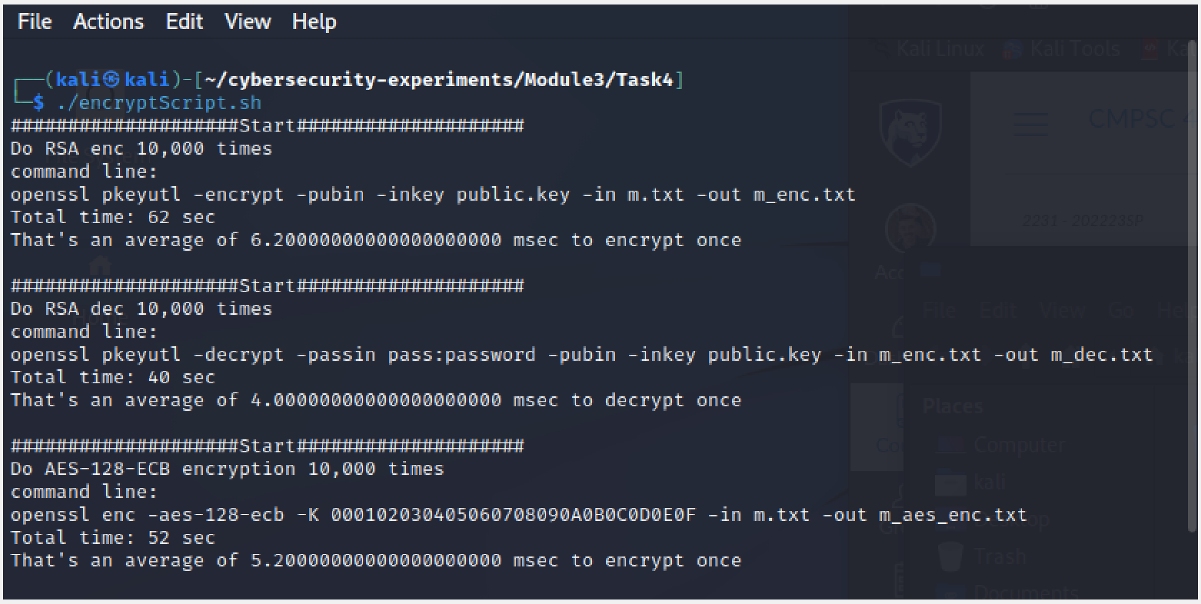
> mentioned whether the CA only needs the servers public key or private key, just its "key". I'm assuming when issuing a certificate based on a Certificate Signing Request and a "key" (in task 2), the CA is including the servers **public** key (that was derived from the private key by the CA in this example) somehow in the certificate. So that when a client decrypts the server's certificate using the CA's certificate it can use the public key to then verify the server's identity?
>
> However, I'm having a hard time finding details online so I will ask you in class. 😃
>
> Regardless, I have no idea why changing a byte in the private key keeps it from being "read". Wouldn't it just be a different valid private key? Is there a mechanism in RSA where only certain keys in the entire span are valid? Unless this webserver uses the public key, that I'm assuming is in the server's certificate, to verify the private key. If it didn't verify, whatever client that connects and verifies the server's certificate using the CA's would be unable to verify the server's identity since it would be unable to decrypt the server's private key encrypted signature.
>
> 3. (c) Please do so, describe and explain your observations
>
> When the domain name is local host, the web browser sees that the certificate provided by the server does not match the domain name. Its like if I gave you an driver's license that didn't have my name on it.

Task 4

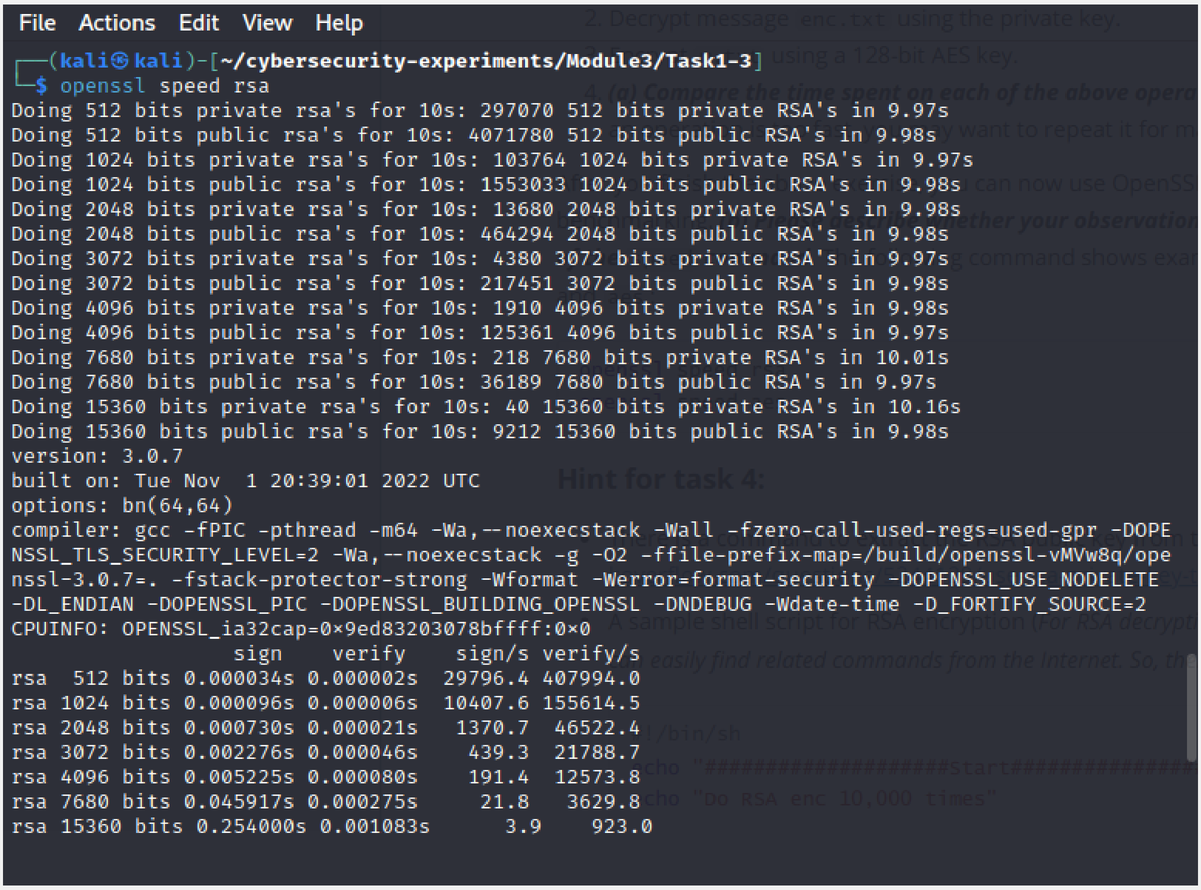
Task 4: Summary

In task 4, we are comparing speed benchmarks between RSA and AES. We do this by writing a bash script that uses openssl encryption and decryption and using openssl's builtin "speed" tool.

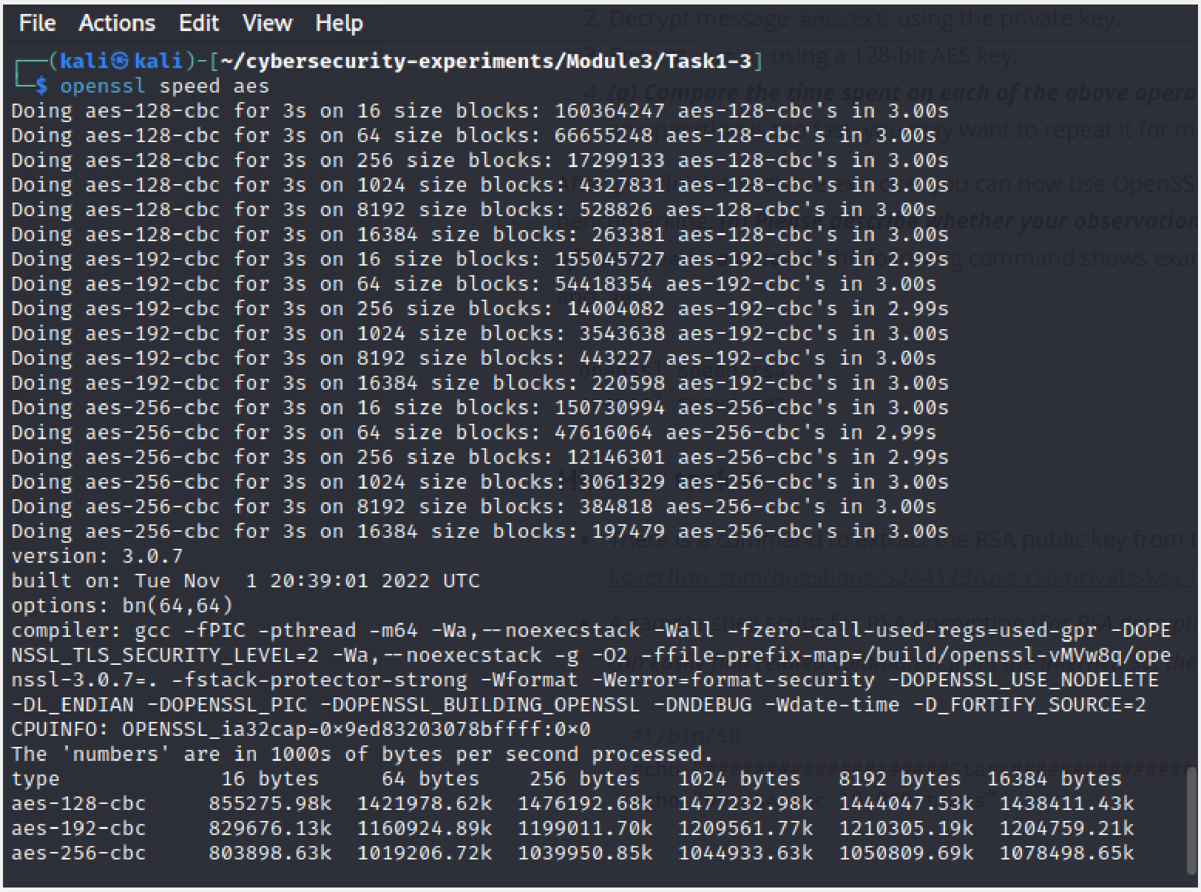
Task 4: Question Answers

> *The screenshot of your operations can be included for better demo.*
>
> 1. (a) Compare the time spent on each of the above operations, and describe your observations.
>
> 
>
>
> ```
> ┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task4]
> └─$ ./encryptScript.sh
> ####################Start####################
> Do RSA enc 10,000 times
> command line:
> openssl pkeyutl -encrypt -pubin -inkey public.key -in m.txt -out m_enc.txt
> Total time: 62 sec
> That's an average of 6.20000000000000000000 msec to encrypt once
>
> ####################Start####################
> Do RSA dec 10,000 times
> command line:
> openssl pkeyutl -decrypt -passin pass:password -pubin -inkey public.key -in m_enc.txt -out m_dec.txt
> Total time: 40 sec
> That's an average of 4.00000000000000000000 msec to decrypt once
>
> ####################Start####################
> Do AES-128-ECB encryption 10,000 times
> command line:
> openssl enc -aes-128-ecb -K 000102030405060708090A0B0C0D0E0F -in m.txt -out m_aes_enc.txt
> Total time: 52 sec
> That's an average of 5.20000000000000000000 msec to encrypt once
> ```
>
> It makes sense that decryption is the fastest. RSA is slower at encrypting. Not sure if the longer key length is a contributing factor. More likely the algorithm.

2. (b) Please describe whether your observations are similar to those from the outputs of the speed command.



```
┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task1-3]
└─$ openssl speed rsa
Doing 512 bits private rsa's for 10s: 297070 512 bits private RSA's in 9.97s
Doing 512 bits public rsa's for 10s: 4071780 512 bits public RSA's in 9.98s
Doing 1024 bits private rsa's for 10s: 103764 1024 bits private RSA's in 9.97s
Doing 1024 bits public rsa's for 10s: 1553033 1024 bits public RSA's in 9.98s
Doing 2048 bits private rsa's for 10s: 13680 2048 bits private RSA's in 9.98s
Doing 2048 bits public rsa's for 10s: 464294 2048 bits public RSA's in 9.98s
Doing 3072 bits private rsa's for 10s: 4380 3072 bits private RSA's in 9.97s
Doing 3072 bits public rsa's for 10s: 217451 3072 bits public RSA's in 9.98s
Doing 4096 bits private rsa's for 10s: 1910 4096 bits private RSA's in 9.98s
Doing 4096 bits public rsa's for 10s: 125361 4096 bits public RSA's in 9.97s
Doing 7680 bits private rsa's for 10s: 218 7680 bits private RSA's in 10.01s
Doing 7680 bits public rsa's for 10s: 36189 7680 bits public RSA's in 9.97s
Doing 15360 bits private rsa's for 10s: 40 15360 bits private RSA's in 10.16s
Doing 15360 bits public rsa's for 10s: 9212 15360 bits public RSA's in 9.98s
version: 3.0.7
built on: Tue Nov  1 20:39:01 2022 UTC
options: bn(64,64)
compiler: gcc -fPIC -pthread -m64 -Wa,--noexecstack -Wall -fzero-call-used-regs=used-gpr -DOPE
NSSL_TLS_SECURITY_LEVEL=2 -Wa,--noexecstack -g -O2 -ffile-prefix-map=/build/openssl-vMVw8q/ope
nssl-3.0.7=. -fstack-protector-strong -Wformat -Werror=format-security -DOPENSSL_USE_NODELETE
-DL_ENDIAN -DOPENSSL_PIC -DOPENSSL_BUILDING_OPENSSL -DNDEBUG -Wdate-time -D_FORTIFY_SOURCE=2
CPUINFO: OPENSSL_ia32cap=0x9ed83203078bffff:0x0
                  sign    verify    sign/s verify/s
rsa  512 bits 0.000034s 0.000002s  29796.4 407994.0
rsa 1024 bits 0.000096s 0.000006s  10407.6 155614.5
rsa 2048 bits 0.000730s 0.000021s   1370.7  46522.4
rsa 3072 bits 0.002276s 0.000046s    439.3  21788.7
rsa 4096 bits 0.005225s 0.000080s    191.4  12573.8
rsa 7680 bits 0.045917s 0.000275s     21.8   3629.8
rsa 15360 bits 0.254000s 0.001083s      3.9    923.0
```



```
┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task1-3]
└─$ openssl speed aes
Doing aes-128-cbc for 3s on 16 size blocks: 160364247 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 64 size blocks: 66655248 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 256 size blocks: 17299133 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 1024 size blocks: 4327831 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 8192 size blocks: 528826 aes-128-cbc's in 3.00s
Doing aes-128-cbc for 3s on 16384 size blocks: 263381 aes-128-cbc's in 3.00s
Doing aes-192-cbc for 3s on 16 size blocks: 155045727 aes-192-cbc's in 2.99s
Doing aes-192-cbc for 3s on 64 size blocks: 54418354 aes-192-cbc's in 3.00s
Doing aes-192-cbc for 3s on 256 size blocks: 14004082 aes-192-cbc's in 2.99s
Doing aes-192-cbc for 3s on 1024 size blocks: 3543638 aes-192-cbc's in 3.00s
Doing aes-192-cbc for 3s on 8192 size blocks: 443227 aes-192-cbc's in 3.00s
Doing aes-192-cbc for 3s on 16384 size blocks: 220598 aes-192-cbc's in 3.00s
Doing aes-256-cbc for 3s on 16 size blocks: 150730994 aes-256-cbc's in 3.00s
Doing aes-256-cbc for 3s on 64 size blocks: 47616064 aes-256-cbc's in 2.99s
Doing aes-256-cbc for 3s on 256 size blocks: 12146301 aes-256-cbc's in 2.99s
Doing aes-256-cbc for 3s on 1024 size blocks: 3061329 aes-256-cbc's in 3.00s
Doing aes-256-cbc for 3s on 8192 size blocks: 384818 aes-256-cbc's in 3.00s
Doing aes-256-cbc for 3s on 16384 size blocks: 197479 aes-256-cbc's in 3.00s
version: 3.0.7
built on: Tue Nov  1 20:39:01 2022 UTC
options: bn(64,64)
compiler: gcc -fPIC -pthread -m64 -Wa,--noexecstack -Wall -fzero-call-used-regs=used-gpr -DOPE
NSSL_TLS_SECURITY_LEVEL=2 -Wa,--noexecstack -g -O2 -ffile-prefix-map=/build/openssl-vMVw8q/ope
nssl-3.0.7=. -fstack-protector-strong -Wformat -Werror=format-security -DOPENSSL_USE_NODELETE
-DL_ENDIAN -DOPENSSL_PIC -DOPENSSL_BUILDING_OPENSSL -DNDEBUG -Wdate-time -D_FORTIFY_SOURCE=2
CPUINFO: OPENSSL_ia32cap=0x9ed83203078bffff:0x0
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes  16384 bytes
aes-128-cbc     855275.98k  1421978.62k  1476192.68k  1477232.98k  1444047.53k  1438411.43k
aes-192-cbc     829676.13k  1160924.89k  1199011.70k  1209561.77k  1210305.19k  1204759.21k
aes-256-cbc     803898.63k  1019206.72k  1039950.85k  1044933.63k  1050809.69k  1078498.65k
```

For RSA-4096, the "speed" tool recorded 29796 encryptions per second and 407994 decryptions per second per 32 bytes. To compare it to AES later, we need to find the time per 16 bytes. Taking the inverse and dividing by 2, we find that it took 0.0168 ms to encrypt and 0.00245 ms to decrypt 16 bytes.

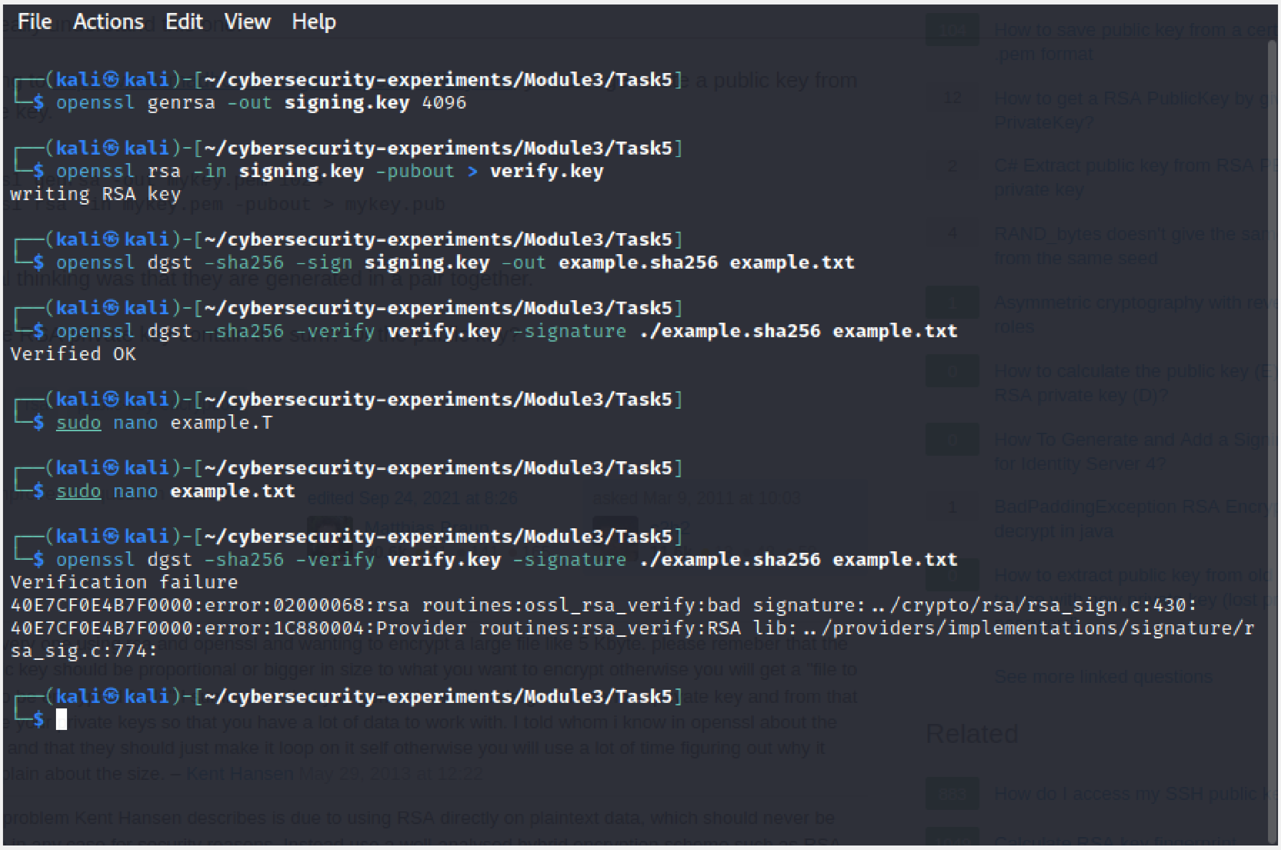
> For AES-128-ECB, 160364247 16 byte sized blocks can be encrypted in 3 seconds. Thats a speed of 0.0000187 ms per 16 byte block.
>
> The speed tool reports much faster times than the bash script. This is due to the latencies in calling a command line tool over and over.

Task 5

Task 5: Summary

In Task 5, we sign a file to create digital signature with a private key. Then we verify the file and signature using the corresponding public key.

Task 5: Question Answers

> *The screenshot of your operations can be included for better demo.*
>
> 
>
>
>
> ```
> ┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task5]
> └─$ openssl genrsa -out signing.key 4096
>
> ┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task5]
> └─$ openssl rsa -in signing.key -pubout > verify.key
> writing RSA key
>
> ┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task5]
> └─$ openssl dgst -sha256 -sign signing.key -out example.sha256 example.txt
>
> ┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task5]
> └─$ openssl dgst -sha256 -verify verify.key -signature ./example.sha256 example.txt
> Verified OK
>
> ┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task5]
> └─$ sudo nano example.T
>
> ┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task5]
> └─$ sudo nano example.txt
>
> ┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task5]
> └─$ openssl dgst -sha256 -verify verify.key -signature ./example.sha256 example.txt
> Verification failure
> 40E7CF0E4B7F0000:error:02000068:rsa routines:ossl_rsa_verify:bad signature:../crypto/rsa/rsa_sign.c:430:
> 40E7CF0E4B7F0000:error:1C880004:Provider routines:rsa_verify:RSA lib:../providers/implementations/signature/r
> sa_sig.c:774:
>
> ┌──(kali㉿kali)-[~/cybersecurity-experiments/Module3/Task5]
> └─$ █
> ```
>
> 1. (a) Please describe how you did the above operations (e.g., what commands do you use, etc.)
>
> I used the "genrsa" and "rsa" tools to create the private and public keys. Then I used the "dgst" or digest tool to create the hash and signature in one command. Only the signature was generated. I then verified the file and signature with the "dgst" again.
>
> 2. (b) Explain your observations.
>
> After changing the original file, verifying using the old signature did not work.
>
> 3. (c) Please also explain why digital signatures are useful.
>
> Digital signatures are useful because they allow us to verify the integrity of a file. If a file was tampered with, the signature would not match when there is an attempt to verify the file.