CSE 5473 — Network Security Autumn 2020

Homework #2

Asymmetric Cryptography *Due Oct 4th, 11:59PM*

1 Diffie-Hellman [15 pts]

Consider a Diffie-Hellman scheme with a common prime q = 11 and a primitive root g = 2.

1. If user A has public key $Y_A = 9$, what is A's private key X_A?

$$\begin{aligned} Y_A &= g^{X_A} \mod q \\ 9 &= 2^{X_A} \mod 11 \\ X_A &= 6 \end{aligned}$$

2. If user B has public key $Y_B = 3$, what is the secret key K shared with A?

$$\begin{aligned} K &= Y_B^{X_A} \mod q \\ &= 3^6 \mod 11 \\ &= 3 \end{aligned}$$

2 RSA [20 pts]

1. Construct a table showing an example of the RSA cryptosystem with parameters p = 17, q = 19, and e = 5. The table should have two rows, one for the plaintext M and the other for the ciphertext C. The columns should correspond to integer values in the range [10; 15] for M. Hint: Write a small program or use a spreadsheet.

```
p = 17
q = 19
n = p * q
e = 5
M = range(10, 16)

for i in M:
   print("M: {}, C: {}".format(i, i ** e % n))

# Output
# M: 10, C: 193
# M: 11, C: 197
# M: 12, C: 122
# M: 13, C: 166
# M: 14, C: 29
# M: 15, C: 2
```

2. In a public-key system using RSA, you intercept the ciphertex C = 10, sent to a user whose public key is e = 5, n = 35. What is the plaintext M?

$$
\begin{gathered}
C = 10, e = 5, n = 35 \\
n = p \times q = 5 \times 7 \\
\varphi(n) = (p-1)(q-1) = (5-1)(7-1) = 4 \times 6 = 24 \\
ed = 1 \mod \varphi(n) \\
ed \mod \varphi(n) = 1 \\
5d \mod 24 = 1 \\
d = 5 (\text{by trial})
\end{gathered}
$$

$$
\begin{aligned}
M &= C^d \mod n, M < n \\
&= 10^5 \mod 35, M < 35 \\
&= 5
\end{aligned}
$$

3. In a public-key system using RSA, the public key of a certain user is e = 31, n = 3599. What is the plaintext M? Hint: you may use the Unix program `factor`[1].

$$
\begin{gathered}
C = 10, e = 31, n = 3599 \\
n = p \times q = 59 \times 61 \\
\varphi(n) = (p-1)(q-1) = (59-1)(61-1) = 58 \times 60 = 3480 \\
ed = 1 \mod \varphi(n) \\
ed \mod \varphi(n) = 1 \\
31d \mod 3480 = 1 \\
d = 3031 (\text{by trial})
\end{gathered}
$$

$$
\begin{aligned}
M &= C^d \mod n, M < n \\
&= C^{3031} \mod 3480, M < 3599
\end{aligned}
$$

4. In a public-key system using RSA, the public key of a certain user with public key e; n leaks his private key d. Being lazy, he re-computes a new e and d using the same n. Is this safe? Why or why not?

   It is not safe to using the same n since $ed \mod \varphi(n) = 1$. Since the e and d were alreay leaked, it is very easy to guess the $\varphi(n)$. By introducing a new factor k, then the equation would be $k(ed) \mod \varphi(n) = 1$. Rewrite the equation, we can get $k(ed)+1 = \varphi(n)$ ($\varphi(n)$ never changed). By changing the factor k, it is possible to get the new e and d value.

[1] http://www.gnu.org/software/coreutils/factor

3 Key Exchange [20 pts]

Tatebayashi, Matsuzaki, and Newman (TMN) proposed the following protocol, which enables Alice and Bob to establish a shared symmetric key K with the help of a trusted server S. Both Alice and Bob know the server's public key K_S. Alice randomly generates a temporary secret K_A, while Bob randomly generates the new key K to be shared with Alice. The protocol then proceeds as follows:

Alice $\Rightarrow$ Server: $K_S\{K_A\}$
Bob $\Rightarrow$ Server: $K_S\{K\}$
Server $\Rightarrow$ Alice: $K \oplus K_A$
Alice recovers key K as $K_A \oplus (K \oplus K_A)$

To summarize, Alice sends her secret to the server encrypted with the server's public key, while Bob sends the newly generated key, also encrypted with the server's public key. The server XORs the two values together and sends the result to Alice. As a result, both Alice and Bob know K. Suppose that evil Charlie eavesdropped on Bob's message to the server. How can he with the help of his equally evil buddy Don, extract the key K that Alice and Bob are using to protect their communications? Assume that Charlie and Don can engage in the TMN protocol with the server, but they do not know the server's private key.

1. Don $\Rightarrow$ Server: $K_S\{K_D\}$
2. Charlie $\Rightarrow$ Server: $K_S\{K\}$ (Replay Bob's message)
3. Server $\Rightarrow$ Don: $K \oplus K_D$
4. Don recovers key K as $K_D \oplus (K \oplus K_D)$

4 Performance Comparison: RSA vs. AES [10 Points]

Asymetric cryptography is typically much slower than symetric cryptography. Please prepare a file (`message.txt`) that contains a 16-byte message. Please also generate an 1024-bit RSA public/private key pair. Then, do the following:

```
# create a 16-byte message
$ openssl rand -hex 16 -out message.txt

# generate a 1024-bit rsa private key
$ openssl genrsa -out priv.pem 1024

# generate a public key
$ openssl rsa -in priv.pem -out pub.pem -pubout -outform PEM
```

1. Encrypt `message.txt` using the public key; save the the output in message `enc.txt`. (Hint: using command `openssl genrsa -des3 -out rsa.key 1024` will generate a public/private key pair stored in file rsa.key, then using `cat message.txt |`

`openssl rsautl -encrypt -inkey rsa.key > message.enc`. If `openssl` is not istalled, please use `sudo apt-get install openssl`)

```
$ time openssl rsautl -encrypt -inkey priv.pem \
-in message.txt -out message.enc

real  0m0.005s
user  0m0.005s
sys   0m0.000s
```

2. Decrypt message `enc.txt` using the private key. (Hint: `cat message.enc | openssl rsautl -decrypt -inkey rsa.key > message.dec`)

```
$ openssl rsautl -decrypt -inkey priv.pem \
-in message.enc -out message.dec
```

3. Encrypt `message.txt` using a 128-bit AES key. (Hint: command such as `openssl enc -aes-128-cbc -e -in msg.txt -out mes.enc -K 00112233445566778899aabbccddeeff -iv 0102030405060708` will perform this job).

4. Compare the time spent on each of the above operations, and describe your observations. If an operation is too fast, you may want to repeat it for many times, and then take an average.

```
$ time openssl enc -aes-128-cbc -e -in message.txt \
-out message_aes.enc -K 00112233445566778899aabbccddeeff \
-iv 0102030405060708

real  0m0.005s
user  0m0.003s
sys   0m0.002s
```

After you finish the above assignment, you can now use OpenSSL's speed command to do such a bench- marking. Please describe whether your observations are similar to those from the outputs of the speed command. The following command shows examples of using speed to benchmark rsa and aes:

```
$ openssl speed rsa
Doing 512 bit private rsa's for 10s: 221247 512 bit private RSA's in 9.74s
Doing 512 bit public rsa's for 10s: 1783693 512 bit public RSA's in 9.84s
Doing 1024 bit private rsa's for 10s: 64119 1024 bit private RSA's in 9.77s
Doing 1024 bit public rsa's for 10s: 686210 1024 bit public RSA's in 9.74s
Doing 2048 bit private rsa's for 10s: 11182 2048 bit private RSA's in 9.88s
Doing 2048 bit public rsa's for 10s: 204876 2048 bit public RSA's in 9.79s
Doing 4096 bit private rsa's for 10s: 1479 4096 bit private RSA's in 9.83s
```

```
Doing 4096 bit public rsa's for 10s: 55703 4096 bit public RSA's in 9.91s
LibreSSL 2.8.3
built on: date not available
options:bn(64,64) rc4(16x,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
                  sign    verify    sign/s verify/s
rsa  512 bits 0.000044s 0.000006s  22725.4 181313.0
rsa 1024 bits 0.000152s 0.000014s   6562.4  70467.9
rsa 2048 bits 0.000883s 0.000048s   1132.1  20937.1
rsa 4096 bits 0.006646s 0.000178s    150.5   5618.8

$ openssl speed aes
Doing aes-128 cbc for 3s on 16 size blocks: 27496672 aes-128 cbc's in 2.85s
Doing aes-128 cbc for 3s on 64 size blocks: 8134544 aes-128 cbc's in 2.96s
Doing aes-128 cbc for 3s on 256 size blocks: 2090208 aes-128 cbc's in 2.97s
Doing aes-128 cbc for 3s on 1024 size blocks: 529811 aes-128 cbc's in 2.98s
Doing aes-128 cbc for 3s on 8192 size blocks: 66300 aes-128 cbc's in 2.97s
Doing aes-192 cbc for 3s on 16 size blocks: 20772495 aes-192 cbc's in 2.55s
Doing aes-192 cbc for 3s on 64 size blocks: 6858203 aes-192 cbc's in 2.98s
Doing aes-192 cbc for 3s on 256 size blocks: 1746481 aes-192 cbc's in 2.98s
Doing aes-192 cbc for 3s on 1024 size blocks: 439997 aes-192 cbc's in 2.98s
Doing aes-192 cbc for 3s on 8192 size blocks: 55329 aes-192 cbc's in 2.97s
Doing aes-256 cbc for 3s on 16 size blocks: 20873161 aes-256 cbc's in 2.88s
Doing aes-256 cbc for 3s on 64 size blocks: 5882779 aes-256 cbc's in 2.98s
Doing aes-256 cbc for 3s on 256 size blocks: 1486542 aes-256 cbc's in 2.98s
Doing aes-256 cbc for 3s on 1024 size blocks: 375080 aes-256 cbc's in 2.98s
Doing aes-256 cbc for 3s on 8192 size blocks: 47083 aes-256 cbc's in 2.98s
LibreSSL 2.8.3
built on: date not available
options:bn(64,64) rc4(16x,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
aes-128 cbc     154568.22k   175698.56k   180165.59k   182233.08k   182708.53k
aes-192 cbc     130450.54k   147427.60k   150241.34k   151440.53k   152565.04k
aes-256 cbc     116079.05k   126497.43k   127891.47k   128936.48k   129608.04k
```

The time were **identical** when we are encrypting the 16-bit file due to the size of payload was too small. However, when we are running the benchmark, the aes is **significantly** faster than rsa.

5 Testing Digital Signatures [15 Points]

Let's use OpenSSL to generate digital signatures. Please prepare a file (example.txt) of any size. Please also prepare an RSA public/private key pair, then do the following:

```
# create example.txt
$ echo "hello fanfan" > example.txt
```

```
# generate a 4096-bit rsa private key
$ openssl genrsa -out priv.pem 4096

# generate a public key
$ openssl rsa -in priv.pem -out pub.pem -pubout -outform PEM
```

1. Sign the SHA256 hash of example.txt; save the output in example.sha256.

   ```
   $ openssl dgst -sha256 -sign priv.pem \
   -out example.sha256 example.txt
   ```

2. Verify the digital signature in example.sha256.

   ```
   $ openssl dgst -sha256 -verify pub.pem \
   -signature example.sha256 example.txt
   Verified OK
   ```

3. Slightly modify example.txt, and verify the digital signature again

   ```
   # overwrite the file
   $ echo "bye fanfan" > example.txt

   # verify the digital signature
   $ openssl dgst -sha256 -verify pub.pem \
   -signature example.sha256 example.txt
   Verification Failure
   ```

Please describe how you did the above operations (e.g., what commands do you use, etc.). Explain your observations. Please also explain why digital signatures are useful.

Only the owner of the private key can sign the file which means the signature is undeniable. Also the verification will be failed when someone chnaged the original content, which protects data integrity. By combining signer's physical identity with their private key, recipients can identify the signer easily and it is impossible for signer to denial the content of they have signed.

In real world, both Microsoft and Apple require software developers to sign their software, otherwise users will experience difficulty to deploy. Also their certificates have to be signed by a trusted **Certificate Authority (CA)** to ensure their identity in real world. In addition, to confirm the time when they have signed their file, CA will provide a time stamp server which endorced the actual time when people signing the file (the time stamp server will use CA's private key to sign the time stamp).

6 Sending emails with public key cryptography [20 Points]

Public Keys is a concept where two keys are involved. One key is a Public Key that can be spread through all sorts of media and may be obtained by anyone. The other key is the Private Key. This key is secret and cannot be spread. This key is only available to the owner. When the system is well implemented the secret key cannot be derived from the public key. Now the sender will crypt the message with the public key belonging to the receiver. Then decryption will be done with the secret key of the receiver.

In this assignment, you will follow detailed instructions and gain some hands on experience of how to use the public key cryptography. To begin with, please follow the manual of GPG on how to create your own keys. Please keep in mind, using public key cryptography to send message will be useful in your whole life. Understanding how to use it is not of wasting of your time.

Also, there are many example tutorials, such as those:

- `http://www.dewinter.com/gnupg_howto/english/GPGMiniHowto.html` or
- `https://help.ubuntu.com/community/GnuPrivacyGuardHowto`. We recommend this link as it contains detailed instructions on how to create your keys, etc.

To install gnupg, you can use:

```
% sudo apt-get install gnupg
```

Assignment Details. This task has to be solved by pairs. First, please try with your peers, and then test with the TA. Specifically:

- Assume you are Bob, please first find another student (say Alice) in the class and ask her to sign her public key with her private key, with the following message (using the instruction manuals for GPG). "`[Alice.num] has the following public key [PK]`", where [Alice.num] is her OSU ID, and PK is her public key.

  Then, Alice sends the signed message (along with the public key) to Bob, and then Bob verifies this is really from Alice by using the public key in the message. So Bob knows this is really from Alice.

- Next, for every student, please find the public key of the TA in the following link: `https://piazza.com/class/ke7xiqs6nk34l6?cid=73`. Use this public key to encrypt the email message sent to TA (his email address is `ma.1189@buckeyemail.osu.edu`). The email message should have the following Subject line: `[CSE 5473] HW2 <Last Name> <First Name>`, with the content of your public key, and signature of the public key (signed with your private key).

  Then the TA will send you an encrypted message with your public key (with some secret content up to the TA), and you need to write it down what you have observed in all these steps, as well as the decrypted message in your report.

```
# generate the key
$ gpg2 --expert --full-gen-key

# list all keys
$ gpg2 --list-keys

# add osu.edu alias for buckeyemail
$ gpg2 --edit-key C468469F767301FB157F34BCD5ACB515B51B1E40
$ adduid
$ save

# import TA's key
$ curl <url_of_the_key> | gpg2 --import

# sign and encrypt my public key
$ gpg2 --armor --export "yao.740@buckeyemail.osu.edu" | gpg2 --armor \
--encrypt --sign --recipient "ma.1189@osu.edu" \
--local-user C468469F767301FB157F34BCD5ACB515B51B1E40

# message received from TA and decrypted
$ cat en_message.txt | openssl base64
hMIDaCFxAqLt7y0SBCMEAUxpgsmuW0SzdPrvIELjauCOMaAE2yJU6yVNcmdc4h/X
R8kPAh4AgorON7w/DY1G6OQaFEfeV/8meIZ6/cTdW2T9AGqnePTuy3lOG1MAZTct
xPtkxF0VWBfgLJ+VGvWSAKJw5DyKExLq4MTLCvuPGQrYnL043X7hhc89pHe5max8
pMfkMOt2KoKsu9JTj2WwZ77YeVO2ubrINKJQWJZkX/XHp78ZpO2j06PmnxFEuAmb
ASMkEdJLAQGLpXaY3e4CidrVHKUneu0E31weZPLe0tUwjDNlhoU925dDSnhd9hkq
kwJ/vXHbh10GMoV/g5I7mkskOOkv5oapEIf7EZtqbDbi

$ cat en_message.txt | gpg2 -d
gpg: encrypted with 521-bit ECDH key, ID 68217102A2EDEF2D, created 2020-09-24
      "Yifan Yao <yao.740@osu.edu>"
congrats Yifan!
```

7 Submiting your report

Please write a report describing how you solve each of the problem above, and submit at CARMEN.