Timothy Jelinek

CIS313, Cryptography
Module 4 Lab – RC4 and AES CTR, Stream Ciphers

In this fourth module, we will explore stream ciphers, specifically the RC4 and AES block cipher in CTR mode, which acts like a stream cipher. RC4 is a stream cipher that is not commonly used anymore and was the subject of several attacks on its implementation in technologies such as WEP. The purpose of the exercise is to familiarize you with the operation of stream ciphers. Modern stream ciphers such as ChaCha20 should be used for any new implementations. Python and other programming language have easy to use libraries that allow you encrypt and decrypt using ChaCha20. See https://pycryptodome.readthedocs.io/en/latest/src/cipher/chacha20.html. Block ciphers also have modes that allow them to act as stream ciphers. In this lab we will explore AES CTR mode. Stream ciphers are often used to encrypt data-in-motion (data being transmitted on a network medium) but have also been used to do things like encrypt Kerberos tickets in Microsoft's Active Directory.

Note: Pay attention to the different tools used to base64 encode and decode the ciphertext. Certutil, base64, openssl base64, and openssl enc -a are all used.

You will be required to submit the following graded items as part of this lab:

- A screenshot and text of your pbkdf generated RC4 key and salt
- A screenshot of a successful RC4 encryption that is base64 encoded
- A screenshot of a successful encryption/decryption process using RC4
- A screenshot, decrypted text/quote, and answered question for provided RC4 ciphertext
- A screenshot and text of your pbkdf generated AES key, IV and salt
- A screenshot of a successful AES encryption that is base64 encoded
- A screenshot, decrypted text/quote, and answered question for provided AES CTR ciphertext

Lab 1 – Generating an RC4 key and Encryption/Decryption

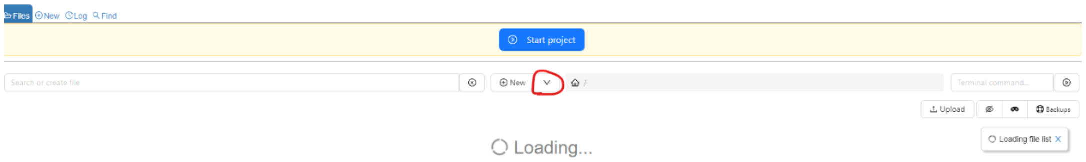
Generating keys for a stream cipher can be done the same two ways (randomly generated or generated using a key derivation function with a password or asymmetric cryptographic value being used as a seed). We will use the PBKDF2 function of OpenSSL to generate keys for this exercise for ease, but feel free to use /dev/random as we did in a previous exercise. The key for rc4 used with OpenSSL should be 128bits long.

If using CoCalc:

- Open your web browser and surf to https://cocalc.com/doc/terminal.html. (Note: If you have access to a Linux OS you can also use that Linux terminal/command line to do this lab.)
- Click on the Blue 'Try a Linux Terminal Now' button.

```
~$ sage
┌────────────────────────────────────────────────────────────────────┐
│ SageMath version 8.7, Release Date: 2019-03-23                     │
│ Create a "Sage Worksheet" file for the notebook interface.         │
│ Enhanced for CoCalc.                                               │
│ Using Python 2.7.15. Type "help()" for help.                       │
└────────────────────────────────────────────────────────────────────┘
sage: 3 + 9
12
sage: factor(12345)
3 * 5 * 823
sage:
Exiting Sage (CPU time 0m0.30s, Wall time 0m21.34s).
~$ lscpu | head
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  2
Core(s) per socket:  2
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
~$ 
```

- Click the blue "Use CoCalc Anonymously" button
- Select 'Projects' from the menu bar under the 'Signed in as' title
- Click the 'Create Project' button. You can leave the default project title as is.
- Click the blue 'Start Project' button.
- Click on the down arrow to the right of "New" and select Terminal



- Accept defaults for name for Terminal file, then click 'Create.'

If using Windows:

- If you are using the Cybersecurity Desktop, open the command line by typing cmd in the Type here to search box on the taskbar. The type the following command: set PATH=%PATH%;C:\Program Files\OpenSSL-Win64\bin

[Animated Example]

1. Next type the following command to generate a RC4 key using a password:

Note: This command works the same between Windows and Linux

CoCalc:

```
openssl enc -rc4 -salt -k YourPassword11 -pbkdf2 -P
```

Windows:

```
openssl enc -rc4 -salt -k YourPassword11 -pbkdf2 -P
```

This command does the following:

- openssl – This is the name of the command we are using. It has many cryptographic functions.

- enc – This tells OpenSSL that we are using the encryption function
- -rc4 – This tells OpenSSL that we will be using the RC4 stream cipher
- -salt – This tells OpenSSL that we want it to generate a random salt to store with the password
- -pbkdf2 – This tells OpenSSL that we want it to use the PBKDF2 function to generate the key
- -k – This is the password we are using as a seed to the PBKDF2 function to create a key. This same key will always be generated every time we use the same function, but it is very hard to find the password if we only have the hexadecimal key.
- -P – This tells OpenSSL not to encrypt but to only print out the salt, key, and IV values.

CoCalc Example:

Animated Example

Windows Example:

Animated Example

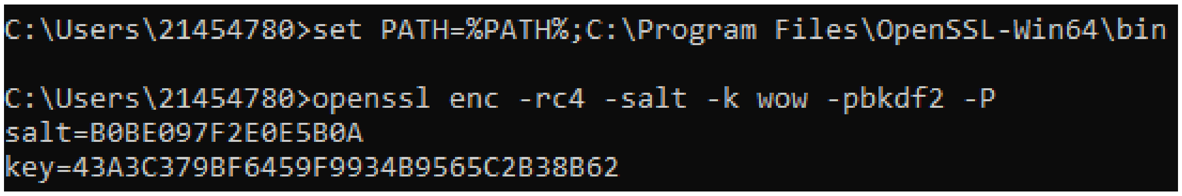
Note: If you want to generate the same key and IV you can include the -S option followed by the salt you previously used.

Record your salt, and key values below

Salt: B0BE097F2E0E580A

Key: 43A3C379BF6459F9934B956C2B38B62

Paste a screenshot of the output of your command below.



```
C:\Users\21454780>set PATH=%PATH%;C:\Program Files\OpenSSL-Win64\bin

C:\Users\21454780>openssl enc -rc4 -salt -k wow -pbkdf2 -P
salt=B0BE097F2E0E5B0A
key=43A3C379BF6459F9934B9565C2B38B62
```

If using CoCalc:

2. Next type the following command to encrypt a message using rc4 and base64 encode the ciphertext

If using CoCalc:

```
echo “Message to Encrypt” | openssl enc -rc4 -e -K <your key> | base64
```

- openssl – This is the name of the command we are using. It has many cryptographic functions.
- enc – This tells OpenSSL that we are using the encryption function
- -rc4 – This tells OpenSSL that we will be using the AES 256 symmetric algorithm in CBC mode
- -e - Specifies that we will be encrypting data
- -K - Allows you to specify a hexadecimal key

- base64

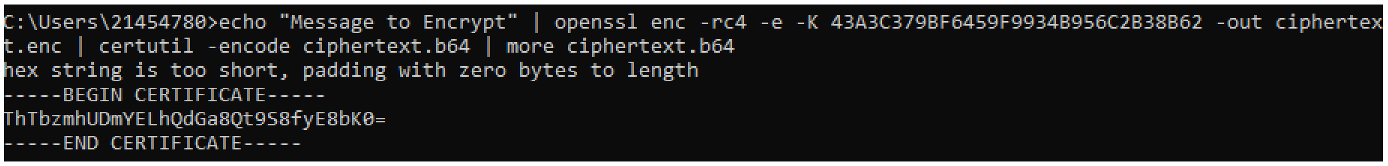
Animated Example

If using Windows:

echo “Message to Encrypt” | openssl enc -rc4 -e -K <your key> -out ciphertext.enc | certutil -encode ciphertext.enc ciphertext.b64 | more ciphertext.b64

- openssl – This is the name of the command we are using. It has many cryptographic functions.
- enc – This tells OpenSSL that we are using the encryption function
- -rc4 – This tells OpenSSL that we will be using the AES 256 symmetric algorithm in CBC mode
- -e - Specifies that we will be encrypting data
- -out - Specifies the destination for the encrypted output file
- -K - Allows you to specify a hexadecimal key

Paste a screenshot of the command being run below:



```
C:\Users\21454780>echo "Message to Encrypt" | openssl enc -rc4 -e -K 43A3C379BF6459F9934B956C2B38B62 -out ciphertex
t.enc | certutil -encode ciphertext.b64 | more ciphertext.b64
hex string is too short, padding with zero bytes to length
-----BEGIN CERTIFICATE-----
ThTbzmhUDmYELhQdGa8Qt9S8fyE8bK0=
-----END CERTIFICATE-----
```

Animated Example

3. In this command we will chain the encryption and decryption functions together. This is essentially what is happening when a stream cipher is used over the network. Running the lab correctly means that the message is encrypted and then decrypted and echoed back to you.

Run the following command:

```
echo “Cryptography is hard.” | openssl enc -rc4 -e -K <your key> |
openssl enc -rc4 -d -K <your key>
```

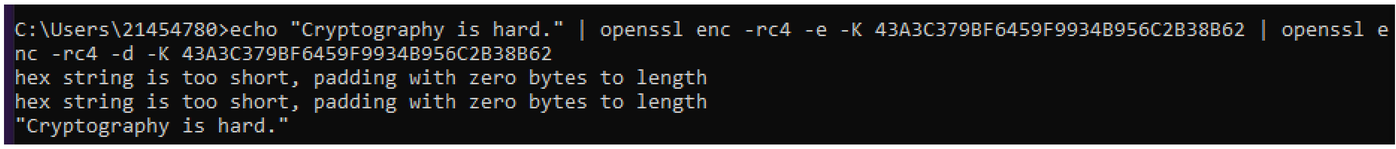
CoCalc Example:

Animated Example

Windows Example:

Animated Example

Paste a screenshot of the command being run below:



```
C:\Users\21454780>echo "Cryptography is hard." | openssl enc -rc4 -e -K 43A3C379BF6459F9934B956C2B38B62 | openssl e
nc -rc4 -d -K 43A3C379BF6459F9934B956C2B38B62
hex string is too short, padding with zero bytes to length
hex string is too short, padding with zero bytes to length
"Cryptography is hard."
```

4. Now it is your turn to be challenged. Decrypt the following message using openssl in rc4 mode using the following key and base64 encoded ciphertext. Hint: look at the last openssl command in the last step for an idea of how to format your decryption command.

Key: ECF42B6E41E12FC7BC97B2C5D0CDC09C

Ciphertext:
39aywxlrvC+kz45EBKjXkgbPEGpRmNXQBATB+lpTd83J6RboHleLSSaqy65sskpfWbnjtan8tFTbbVbCQLDHdx+VrT+Z3kbFAFcBBfqgBw+GanwpGd/sWYT6wE3QkSyRIRyx1QYVWCbCjAIUEgAWZsMRLf9XZz2365HjRnN1sPRs/DMuFFzYSubEzE+wfesdKBFviAZwTBuCJNByXAmHBVyRUBPLW6J2H1tDzqUREWoZEuRFJQrQACgNtJBhuA/kBA==

Pipe the echo command into OpenSSL one of the following ways:

<u>Note: Use quotes around the ciphertext in CoCalc and no quotes in Windows</u>

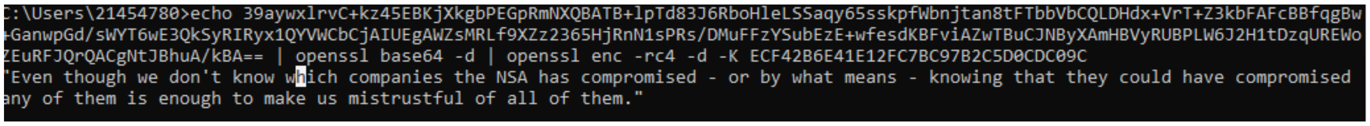
```
echo <ciphertext> | openssl  decrypt command (add -a tag to decode
base64)

echo <ciphertext> | openssl base64 -d | openssl decrypt command (don't'
add the -a tag)
```

Text of the quote here:
"Even though we don't know which companies the NSA has compromised – or by what means – knowing that they could have compromised any of them is enough to make us mistrustful of all of them."

Who said this quote:
Bruce Schneier

Screenshot of the command being run on your terminal:



```
:\Users\21454780>echo 39aywxlrvC+kz45EBKjXkgbPEGpRmNXQBATB+lpTd83J6RboHleLSSaqy65sskpfWbnjtan8tFTbbVbCQLDHdx+VrT+Z3kbFAFcBBfqgBw
+GanwpGd/sWYT6wE3QkSyRIRyx1QYVWCbCjAIUEgAWZsMRLf9XZz2365HjRnN1sPRs/DMuFFzYSubEzE+wfesdKBFviAZwTBuCJNByXAmHBVyRUBPLW6J2H1tDzqUREWo
ZEuRFJQrQACgNtJBhuA/kBA== | openssl base64 -d | openssl enc -rc4 -d -K ECF42B6E41E12FC7BC97B2C5D0CDC09C
"Even though we don't know which companies the NSA has compromised - or by what means - knowing that they could have compromised
any of them is enough to make us mistrustful of all of them."
```

<u>Lab 2 – Generating an AES 256 key and CTR mode Encryption/Decryption</u>

Generating keys for a stream cipher can be done the same two ways (randomly generated or generated using a key derivation function with a password or asymmetric cryptographic value being used as a seed). We will use the PBKDF2 function of OpenSSL to generate keys for this exercise for ease, but feel free to use /dev/random as we did in a previous exercise. The key for AES 256 CTR used with OpenSSL should be 256 bits long. You will notice that AES in CTR mode uses an IV while RC4 does not.

5. Type the following command to generate an AES 256 key and IV using a password:

Note: This command works the same between Windows and Linux

CoCalc:

```
openssl enc -aes-256-ctr -salt -k YourPassword11 -pbkdf2 -P
```

Windows:

```
openssl enc -aes-256-ctr -salt -k YourPassword11 -pbkdf2 -P
```

This command does the following:

openssl – This is the name of the command we are using. It has many cryptographic functions.

enc – This tells OpenSSL that we are using the encryption function

-aes-256-ctr – This tells OpenSSL that we will be using the AES 256 in CTR mode

-salt – This tells OpenSSL that we want it to generate a random salt to store with the password

-pbkdf2 – This tells OpenSSL that we want it to use the PBKDF2 function to generate the key

-k – This is the password we are using as a seed to the PBKDF2 function to create a key. This same key will always be generated every time we use the same function, but it is very hard to find the password if we only have the hexadecimal key.

-P – This tells OpenSSL not to encrypt but to only print out the salt, key, and IV values.

CoCalc Example:

Animated Example

Windows Example:

Animated Example

Note: If you want to generate the same key and IV you can include the -S option followed by the salt you previously used.

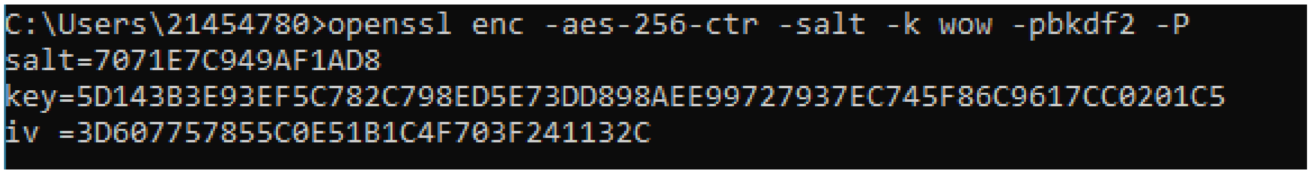
Record your salt, and key values below

Salt: 7071E7C949AF1AD8

Key: 5D143B3E93EF5C782C798ED5E73DD898AEE99727937EC745F86C9617CC0201C5

IV: 3D607757855C0E51B1C4F703F241132C

Paste a screenshot of the output of your command below.



```
C:\Users\21454780>openssl enc -aes-256-ctr -salt -k wow -pbkdf2 -P
salt=7071E7C949AF1AD8
key=5D143B3E93EF5C782C798ED5E73DD898AEE99727937EC745F86C9617CC0201C5
iv =3D607757855C0E51B1C4F703F241132C
```

6. Next type the following command to encrypt a message using aes-256-ctr and base64 encode the ciphertext

CoCalc:

```
echo “May the force be with you.” | openssl enc -aes-256-ctr -e -K
<your key> -iv <your iv> -a
```

Windows:

```
echo “May the force be with you.” | openssl enc -aes-256-ctr -e -K
<your key> -iv <your iv> -a
```

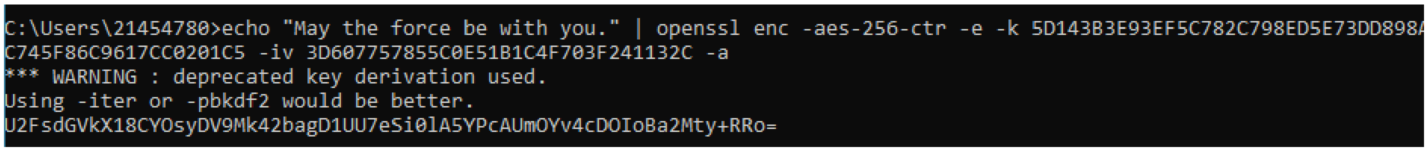
CoCalc Example:

[Animated Example]

Windows Example:

[Animated Example]

Paste a screenshot of the command being run:



```
C:\Users\21454780>echo "May the force be with you." | openssl enc -aes-256-ctr -e -k 5D143B3E93EF5C782C798ED5E73DD898A
C745F86C9617CC0201C5 -iv 3D607757855C0E51B1C4F703F241132C -a
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
U2FsdGVkX18CYOsyDV9Mk42bagD1UU7eSi0lA5YPcAUmOYv4cDOIoBa2Mty+RRo=
```

7. Now it is your turn to be challenged. Decrypt the following message using openssl in aes-ctr-mode using the following key and base64 encoded ciphertext. Hint: look at the last openssl command in the last step for an idea of how to format your decryption command.

Key: 51A7F88557E84B370FA3483D73DF1246DA33BC714555AE47990E63F72B523959

IV: 3C78EE9090114DE8FCB300BADCA25047

Ciphertext:
UnKaB7CSzfcZaE0fdE9H5YAVaDaTDpppcRTigjCxCppoQ9TI2oZsNnsWo9KqSWuEBzDQixOXviv+lsgA8g
x5mrd+qs0yw4MU08kc38/eC1c1bVhb0bJ5kqah6A+riwvjMiAafsjmINH348Kbow7G9qeg2jJOB4qw3dY
opM7Vj2cnKJqjlYYlAOv9QlTTsh47UPTaozeLRJq0zhs=

Pipe the echo command into OpenSSL one of the following ways:

<u>Note: Use quotes around the ciphertext in CoCalc and no quotes in Windows</u>

```
echo <ciphertext> | openssl  decrypt command (add -a tag to decode
base64)

echo <ciphertext> | openssl base64 -d | openssl decrypt command (don’t’
add the -a tag)
```

Text of the quote here:

I couldn’t figure this one out.

Who said this quote:

Screenshot of the command being run on your terminal: