Lab 1: Cryptography

Christian O'meara
Willow Fussell
Elhacen Elmoustapha
Noureldin Youssef
CSE 566

Abstract

This lab introduces students to various cryptographic concepts and tools using the Kali Linux environment provided by UofL CSE’s CyberPVE. Students will explore symmetric encryption, comparing different ciphers and modes (ECB vs. CBC) using the openssl software, and examine the effects of corrupted ciphertext. Additionally, the lab covers generating message digests and HMACs, with tasks to understand key size requirements and hash function properties. Students will also act as a Certificate Authority (CA), creating and issuing digital certificates. Finally, the lab includes password-based authentication exercises, involving hashing, salting, and cracking password hashes, as well as an offline analysis using password cracking tools. The goal is to deepen understanding of cryptographic methods and their practical applications in information security.

Part 1: Symmetric encryption using different ciphers and modes

In Lab 1, we experimented with several encryption and decryption symmetric ciphers using different cipher modes. We specifically worked with examples of AES(Advanced Encryption Standard) and DES(Data Encryption Standard) ciphers. Some of these ciphers require keys and initialization vectors (IVs) of specific lengths, while others do not require IVs, depending on their mode of operation. For instance, in the following encryption and decryption commands, one does not use an IV because it employs the ECB (Electronic Codebook) mode.We encrypted and decrypted multiples file. Each text file is named name_plain.txt where the name is the cipher type. Similarly, the decrypted file are named name_decrypted.bin where the name is the cipher type used to decrypt the txt file.

Each txt file contains the following text.

“encrypting and decrypting using [name of the cipher type]”

For example, “encrypting and decrypting using aes 128 cbc.”

AES

-aes-128-cbc,

-aes-128-cfb,

aes-128-ecb,

Aes-192-cbc

DES

Des-ede3-ecb

Des-ofb

openssl enc -aes-128-cbc -e -in aes_128_cbc_plain.txt -out aes_128_cbc.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708000000000000000

$ openssl enc -aes-128-cbc -d -in aes_128_cbc.bin -out aes_128_cbc_decrypted.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708000000000000000

-aes-128-cfb,

openssl enc -aes-128-cfb -e -in aes_128_cfb_plain.txt -out aes_128_cfb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708000000000000000

openssl enc -aes-128-cfb -d -in aes_128_cfb.bin -out aes_128_cfb_decrypted.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708000000000000000

aes-128-ecb,

Does not require IV

openssl enc -aes-128-ecb -e -in aes_128_ecb_plain.txt -out aes_128_ecb.bin -K 00112233445566778889aabbccddeeff

openssl enc -aes-128-ecb -d -in aes_128_ecb.bin -out aes_128_ecb_decrypted.txt -K 00112233445566778889aabbccddeeff

Aes-192-cbc

openssl enc -aes-192-cbc -e -in aes_192_cbc_plain.txt -out aes_192_cbc.bin -K 00112233445566778899aabbccddeeff0011223344556677 -iv 0102030405060708000000000000000

openssl enc -aes-192-cbc -d -in aes_192_cbc.bin -out aes_192_cbc_decrypted.txt -K 00112233445566778899aabbccddeeff0011223344556677 -iv 0102030405060708000000000000000

Des-ede3-ecb

Does not require IV

openssl enc -des-ede3-ecb -e -in des_ede3_ecb_plain.txt -out des_ede3_ecb.bin -K 00112233445566778899aabbccddeeff0011223344556677

openssl enc -des-ede3-ecb -d -in des_ede3_ecb.bin -out des_ede3_ecb_decrypted.txt -K 00112233445566778899aabbccddeeff0011223344556677

openssl enc -des-ofb -e -in des_ofb_plain.txt -out des_ofb.bin -K 0011223344556677 -iv 0102030405060708

openssl enc -des-ofb -d -in des_ofb.bin -out des_ofb_decrypted.txt -K 0011223344556677 -iv 0102030405060708

Openssl speed

The speed of computation is an important factor for any cryptographic operation. OpenSSL includes a built-in benchmarking tool that can help in assessing cryptographic algorithms, including both encryption and decryption operations. This benchmark can be invoked using the openssl speed command. In this lab, we analyzed the performance of symmetric encryption and decryption ciphers of type DES and AES.It’s worth mentioning that `openssl speed` command traditionally defaults to testing cryptographic algorithms in modes like CBC because these modes are commonly used and standardized.

The following table indicates the performance of different ciphers using openssl speed command. We executed commands to measure speed of different AES/DES ciphers. The result actual output can be views in the following text files, AES_performance.txt and DES_performance.txt

AES ciphers: performance

Command:openssl speed aes

```
version: 3.0.11
built on: Tue Sep 19 16:58:30 2023 UTC
options: bn(64,64)
compiler: gcc -fPIC -pthread -m64 -Wa,--noexecstack -Wall -fzero-call-used-regs=used-gpr -
 DOPENSSL_TLS_SECURITY_LEVEL=2 -Wa,--noexecstack -g -O2 -ffile-prefix-map=/build/reproducible-
 path/openssl-3.0.11=. -fstack-protector-strong -fstack-clash-protection -Wformat -Werror=format-
 security -fcf-protection -DOPENSSL_USE_NODELETE -DL_ENDIAN -DOPENSSL_PIC -
 DOPENSSL_BUILDING_OPENSSL -DNDEBUG -Wdate-time -D_FORTIFY_SOURCE=2
CPUINFO: OPENSSL_ia32cap=0×80202001479bffff:0×0
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      65881.55k    98181.78k   111500.63k   254759.59k   262561.79k   259276.80k
aes-192-cbc      59289.37k    83033.79k    91535.87k   211962.88k   218606.25k   211779.58k
aes-256-cbc      53442.17k    73111.95k    80284.07k   187151.36k   191466.15k   191457.96k
```

DES ciphers: performance

Command:openssl speed des

```
version: 3.0.11
built on: Tue Sep 19 16:58:30 2023 UTC
options: bn(64,64)
compiler: gcc -fPIC -pthread -m64 -Wa,--noexecstack -Wall -fzero-call-used-regs=used-gpr -
 DOPENSSL_TLS_SECURITY_LEVEL=2 -Wa,--noexecstack -g -O2 -ffile-prefix-map=/build/reproducible-
 path/openssl-3.0.11=. -fstack-protector-strong -fstack-clash-protection -Wformat -
 Werror=format-security -fcf-protection -DOPENSSL_USE_NODELETE -DL_ENDIAN -DOPENSSL_PIC -
 DOPENSSL_BUILDING_OPENSSL -DNDEBUG -Wdate-time -D_FORTIFY_SOURCE=2
CPUINFO: OPENSSL_ia32cap=0×80202001479bffff:0×0
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes  16384 bytes
des-cbc          42604.34k    51741.14k    54792.36k    55648.26k    55683.75k    56164.35k
des-ede3         17914.75k    19509.29k    19956.57k    20188.84k    20231.51k    20288.85k
```

Types: Indicates the type of AES/DES operation/cipher and mode being tested.For example, aes-128 cbc represents AES/DES encryption and decryption using a 128-bit key in CBC (Cipher Block Chaining) mode.

Block Sizes: The columns represent different block sizes, ranging from 16 bytes to 8192 bytes. Block size indicates the size of data blocks that AES/DES processes in a one encryption/ decryption operation.

Throughput: The numbers in the table represent the throughput of AES/DES operations in thousands of bytes per second (k). This indicates how many bytes of data AES/DES can process per second for the given key size, block size, and mode of operation. Higher numbers indicate faster throughput.

Implications:

AES significantly outperforms DES in both speed and security. AES processes data much faster than DES, especially for larger block sizes, and offers stronger security with key sizes of 128, 192, and 256 bits, compared to DES's 56-bit key, which is now considered insecure. Additionally, 3DES, while more secure than DES, is still slower and less efficient than AES.

Therefore, AES is the preferred choice for modern encryption needs due to its superior efficiency and security.

Part 2: Encryption Mode – ECB vs. CBC

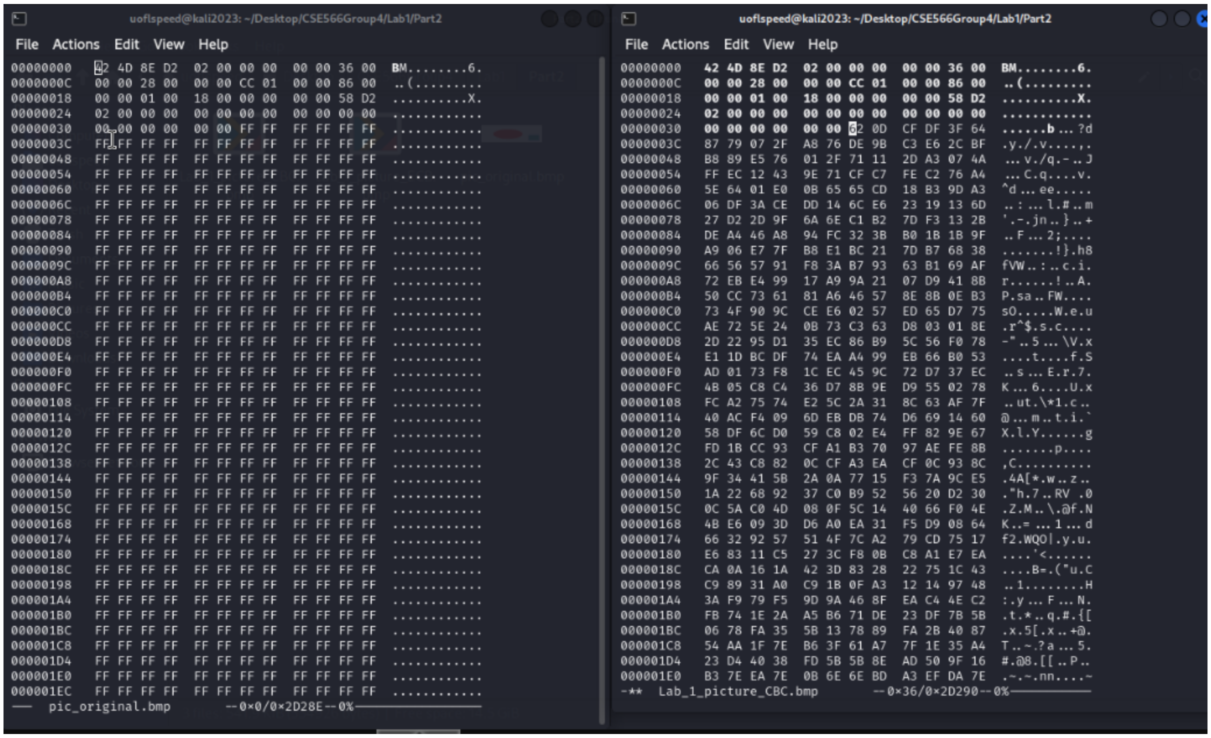
ECB mode encrypts each block of data independently using the same key. Consequently, if two blocks of plaintext are identical, they will produce identical ciphertext blocks. This can create patterns in the ciphertext, which attackers can exploit. Additionally, ECB mode lacks diffusion, so small changes in the plaintext lead to predictable changes in the ciphertext.

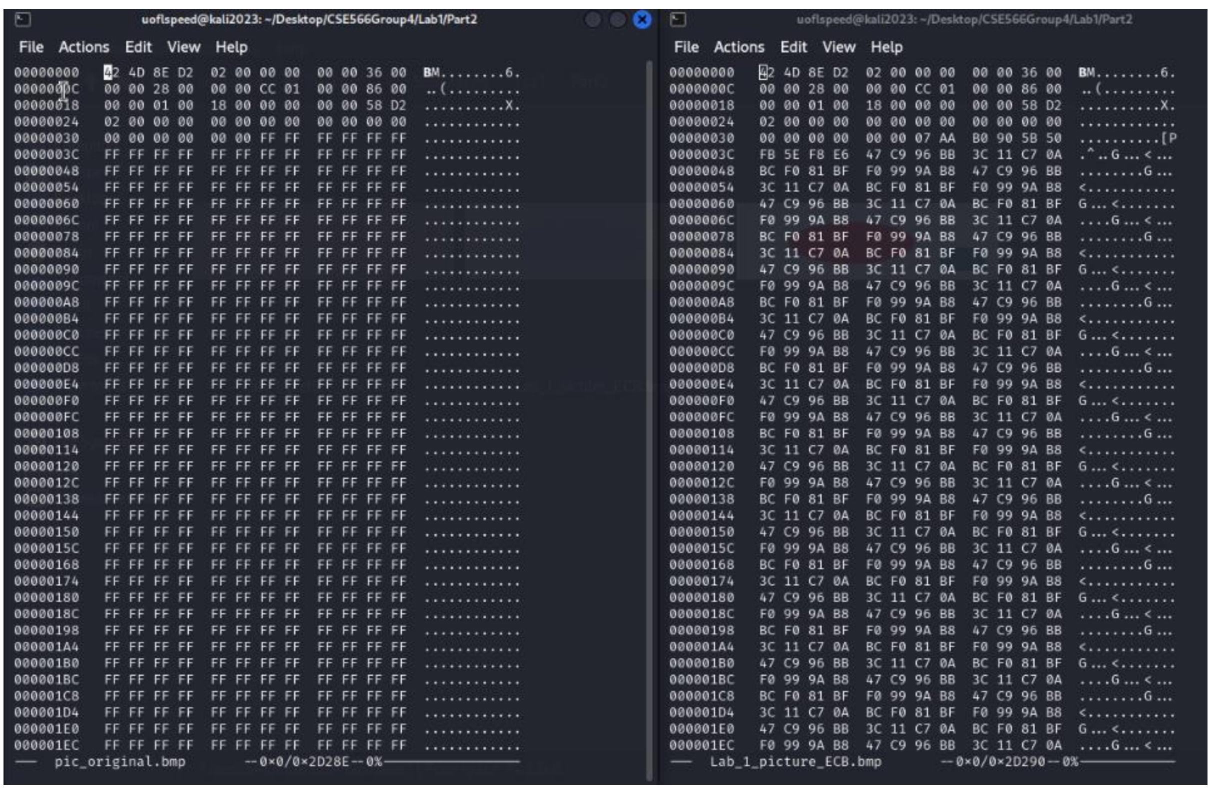
In contrast, CBC mode XORs each plaintext block with the previous ciphertext block before encryption. This ensures that even if plaintext blocks are identical, the resulting ciphertext blocks will differ due to the chaining effect. CBC mode enhances security by eliminating patterns in the ciphertext and providing diffusion, making it more difficult for attackers to analyze and exploit.

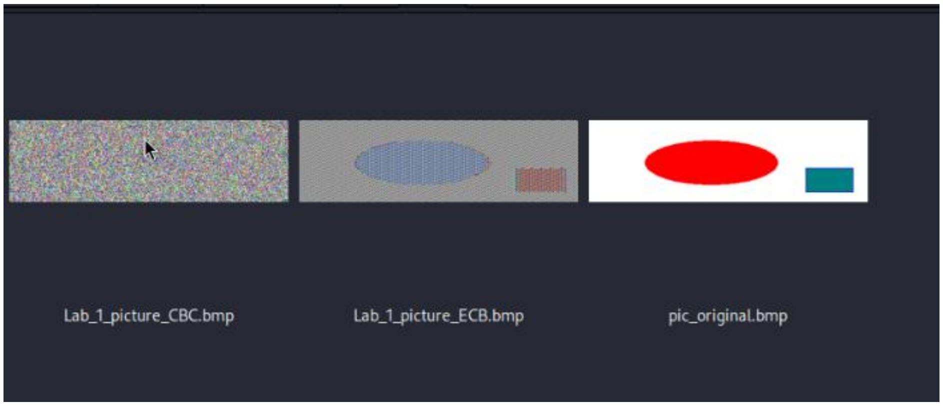
When choosing between ECB and CBC mode, always opt for CBC. ECB mode reveals information about the plaintext because identical plaintext blocks result in identical ciphertext blocks. CBC mode prevents this, making ECB mode insecure and unsuitable for use.

openssl enc -aes-128-cbc -e -in pic_original.bmp -out Lab_1_picture_CBC.bmp -K 00112233445566778889aabbccddeeff -iv 0102030405060708000000000000000

openssl enc -aes-128-ecb -e -in pic_original.bmp -out aes_128_ECB.bmp -K 00112233445566778889aabbccddeeff


uoflspeed@kali2023: ~/Desktop/CSE566Group4/Lab1/Part2
File Actions Edit View Help
pic_original.bmp --0×0/0×2D28E--0%
uoflspeed@kali2023: ~/Desktop/CSE566Group4/Lab1/Part2
File Actions Edit View Help
-** Lab_1_picture_CBC.bmp --0×36/0×2D290--0%


uoflspeed@kali2023: ~/Desktop/CSE566Group4/Lab1/Part2
File Actions Edit View Help
pic_original.bmp --0×0/0×2D28E--0%
uoflspeed@kali2023: ~/Desktop/CSE566Group4/Lab1/Part2
File Actions Edit View Help
Lab_1_picture_ECB.bmp --0×0/0×2D290--0%



The ECB encrypted image reveals patterns that can give away some information about the original image, making it less secure. This is because ECB mode encrypts each block of data independently without any interdependency. Therefore, identical plaintext/image blocks result in identical ciphertext/image blocks, leading to the visibility of repeating patterns.

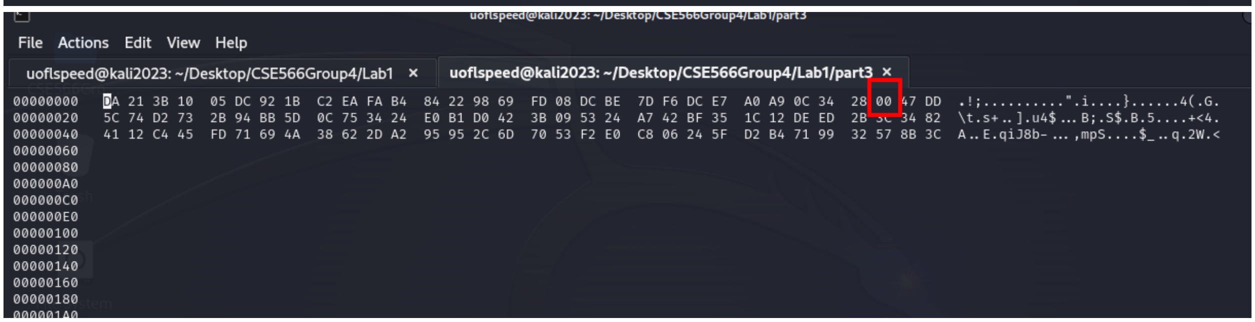
In contrast, the CBC encrypted image appears completely random and does not reveal any useful information about the original image, making it more secure. This is because CBC mode uses an initialization vector (IV) and each block of the image data is combined with the previous encrypted block before being encrypted itself.

Part 3: Encryption Mode – Corrupted Cipher Text

How much information can be recovered by decrypting a file where one of the bytes in the encrypted file has been corrupted?

The amount of information that can be recovered from a decrypted file with one byte corrupted will depend on the encryption method used

- **Block Cipher in Electronic Codebook (ECB) Mode**: In ECB mode, each block of data is encrypted independently. If a byte in an encrypted block is corrupted, it typically affects only its output during decryption.
- **Block Cipher in Cipher Block Chaining (CBC) Mode**: In CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. Therefore, a single byte corruption in one block affects its decryption and corrupts the beginning of the next block (due to the chaining).
- **Stream Ciphers**: For stream ciphers, the plaintext is combined with a pseudorandom cipher digit stream (keystream) using bitwise operations such as XOR. Corruption in one byte of the ciphertext will corrupt exactly one byte of the plaintext upon decryption. Thus, all information except the corrupted byte can be perfectly recovered.
- **Authenticated Encryption Modes (e.g., GCM, CCM)**: These modes provide both encryption and integrity checking. If a byte is corrupted, the integrity check (authentication tag) will typically fail during decryption. This means that the decryption process will either refuse to output any plaintext or will signal that the data could not be authenticated, depending on the specific mode and implementation. In practice, this means no information is reliably recovered without additional error-correcting measures.



In your lab write up reflect on the answer you gave before conducting the exercise, was your thinking correct? Why or why not? What are some implications of what you learned as a result of doing the activity?

I hypothesized that a corrupted byte in the encrypted file would primarily affect only the block containing that byte.

Results

Decrypting the corrupted AES-128-CBC encrypted file showed that not only the block with the corrupted byte was affected but also the subsequent block. This occurs because, in CBC mode, each block is XORed with the previous ciphertext block, so corruption propagates to at least one additional block.

Analysis of Results

My initial prediction underestimated the impact of corruption in CBC mode. The results revealed significant error propagation affecting multiple blocks, which was not initially anticipated.

Part 4: Generating Message Digest and MAC

We used the following three one-way:
SHA-256, MD5, SHA-1

```
┌──(uoflspeed㉿kali2023)-[~/Desktop/CSE566Group4/Lab1/part4]
└─$ openssl dgst -md5 testfile.txt
MD5(testfile.txt)= e9fe0163fd69b80b738391145a817305

┌──(uoflspeed㉿kali2023)-[~/Desktop/CSE566Group4/Lab1/part4]
└─$ openssl dgst -sha1 testfile.txt
SHA1(testfile.txt)= 6d6d917e54e3390a680129c8eaec372db6440711

┌──(uoflspeed㉿kali2023)-[~/Desktop/CSE566Group4/Lab1/part4]
└─$ openssl dgst -sha256 testfile.txt
SHA2-256(testfile.txt)= ae6ce50d69fee27fe70ba5f040a392f16b89e928a6a1cfc387dd2fdd2e96609b
```

```
┌──(uoflspeed㉿kali2023)-[~/Desktop/CSE566Group4/Lab1/part4]
└─$ openssl dgst -md5 -hmac "simplekey" testfile.txt
HMAC-MD5(testfile.txt)= 45faa8bb425199742a375b9124aafb97

┌──(uoflspeed㉿kali2023)-[~/Desktop/CSE566Group4/Lab1/part4]
└─$ openssl dgst -sha256 -hmac "simplekey" testfile.txt
HMAC-SHA2-256(testfile.txt)= 4953846c760130e3a0b7142400dadbbfc61909d0af027b4e0b5142b1949e0b08

┌──(uoflspeed㉿kali2023)-[~/Desktop/CSE566Group4/Lab1/part4]
└─$ openssl dgst -sha1 -hmac "longerkeyexample" testfile.txt
HMAC-SHA1(testfile.txt)= 0ccb50c00c7278acd40ffe8bb516888ad10ffc79
```

Answer the following questions in the lab report:

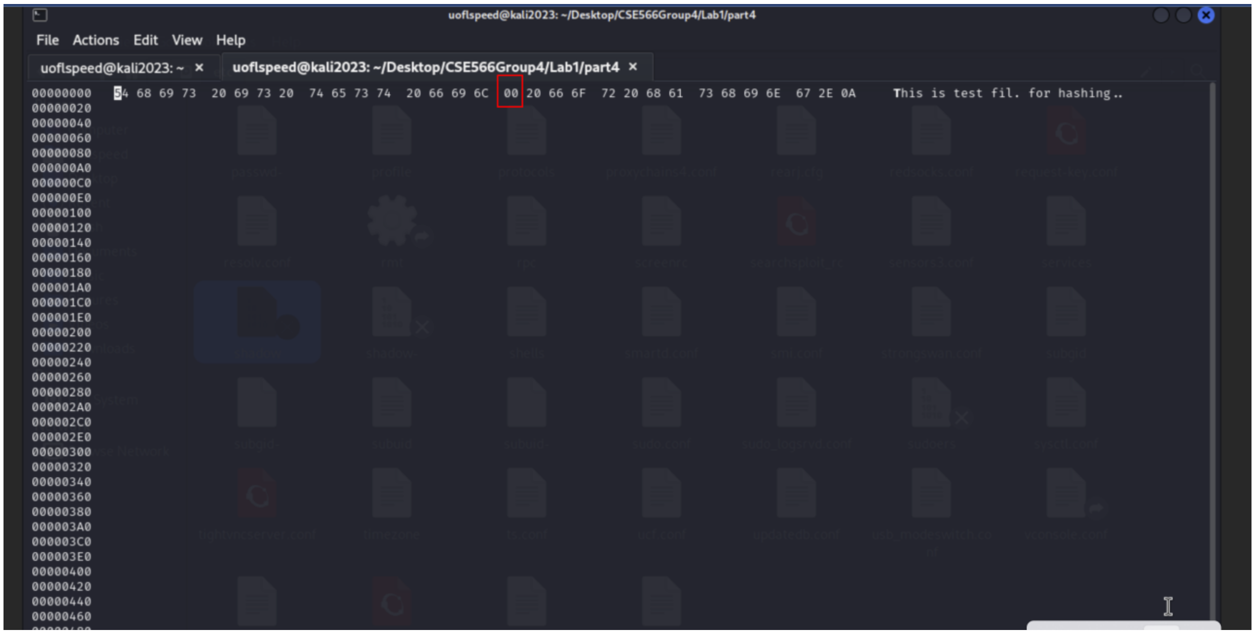
Do we have to use a key with a fixed size in HMAC? If so, what is the key size? If not, why?

- No, the key in HMAC does not need to be of a fixed size. HMAC processes keys internally to fit the block size of the hash function. If the key is longer than the hash block size, it is hashed to shorten it. If it's shorter, it is padded.
- Experiment with keys of varying lengths to see the effect on the HMAC output.

How dissimilar are they? Write a short program (any language) to count how many bits are different between H1 and H2. In your lab report include H1, H2, your code (or at least the main logic part of it), and the output of the program.

```python
def count_diff_bits(hex1, hex2):
    # Convert hex to binary
    bin1 = bin(int(hex1, 16))[2:].zfill(128)  # Ensuring full length
    bin2 = bin(int(hex2, 16))[2:].zfill(128)

    # Count differences
    diff = sum(b1 != b2 for b1, b2 in zip(bin1, bin2))
    return diff

# Read hash values from files
with open('hash1.txt', 'r') as file:
    hash1 = file.read().strip()

with open('hash2.txt', 'r') as file:
    hash2 = file.read().strip()

# Calculate and print the number of differing bits
difference = count_diff_bits(hash1, hash2)
print(f"Differences in bits: {difference}")
```

Editing with hex editor



Flipping one bit in the input file resulted in 59 bits being different between the two MD5 hash outputs, H1 and H2. This demonstrates the huge effect, where a minimal change in the input causes changes in the output hash, ensuring that even small changes are detectable.

Part 5: Become a Certificate Authority (CA)

Using the command:

```
cp /usr/lib/ssl/openssl.cnf ./openssl.cnf
```

We were able to copy the default OpenSSL configuration file to our working directory. The following directories were then created: demoCA/certs, demoCA/crl, demoCA/newcerts, and demoCA/private, along with the files demoCA/index.txt and demoCA/serial with an initialized value of 1000.

The openssl.cnf file was then modified to ensure the required fields were included. After the preparation, the next step was to generate a self-signed certificate for the CA. We began with the command:

```
openssl req -new -x509 -days 3650 -key demoCA/private/ca.key -out
demoCA/certs/ca.crt -config openssl.cnf -subj
'/CN=www.pkilabserver.com/O=pkilabserver/C=US/ST=Kentucky'
```

And the certificate was verified using:

```
openssl x509 -noout -text -in demoCA/certs/ca.crt
```

Following this, we created a certificate for the server “pkilabserver.com” by generating the public/private key pair for the server using:

```
openssl genpkey -algorithm RSA -out server.key -aes256
```

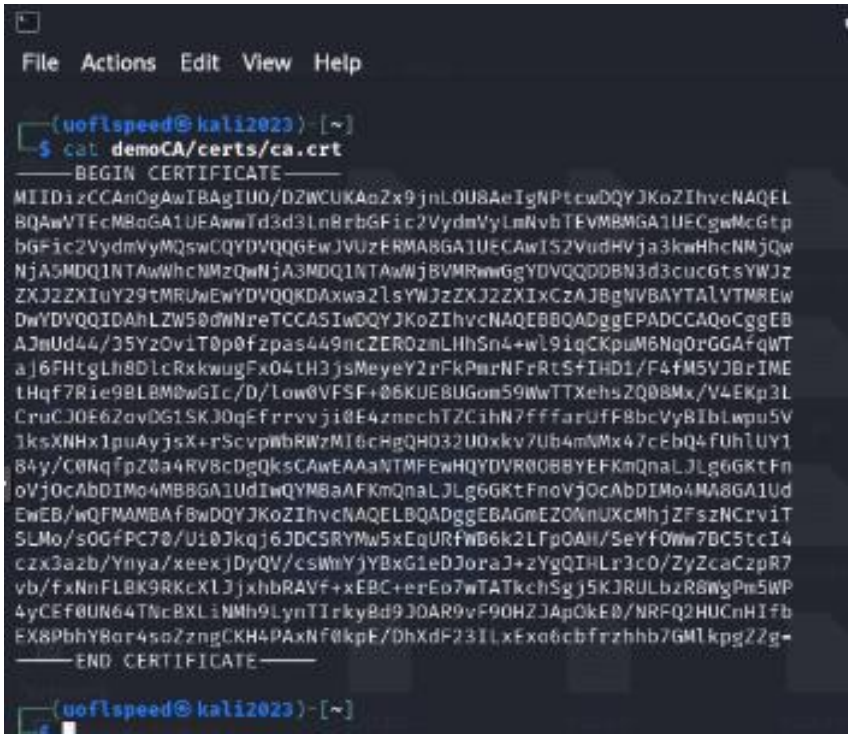
And creating a CSR using:

```
openssl req -new -key server.key -out server.csr -config openssl.cnf -subj
'/CN=www.pkilabserver.com/O=pkilabserver/OU=IT
Department/C=US/ST=Kentucky/L=Louisville'
```

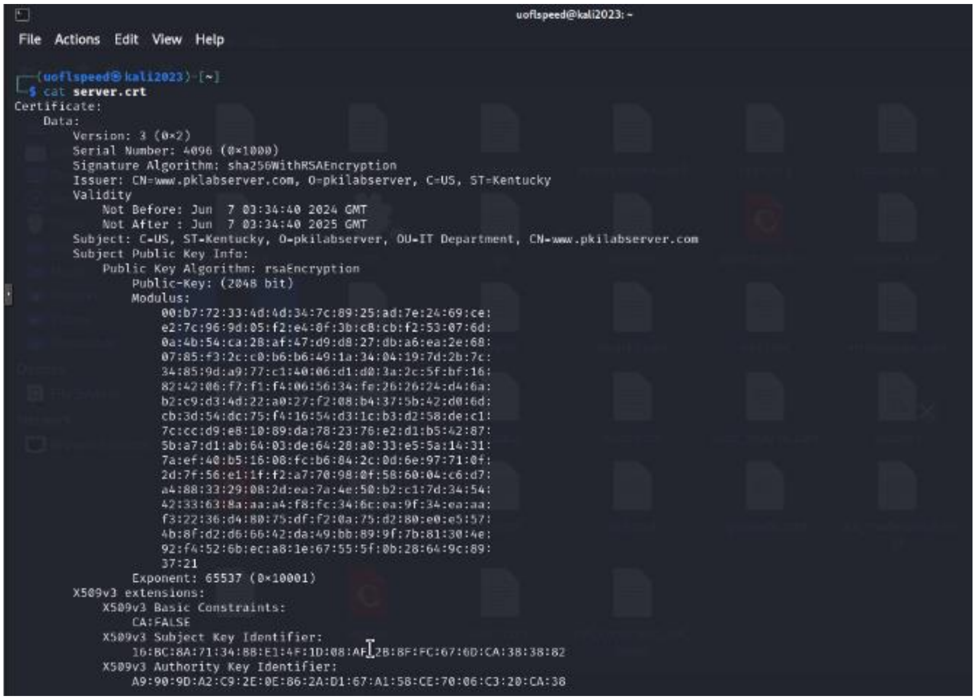
The CSR was then signed with the CA’s certificate using:

```
openssl ca -in server.csr -out server.crt -cert demoCA/certs/ca.crt -keyfile
demoCA/private/ca.key -config openssl.cnf
```

Public Key of CA:



Details of Signed Certifcate:



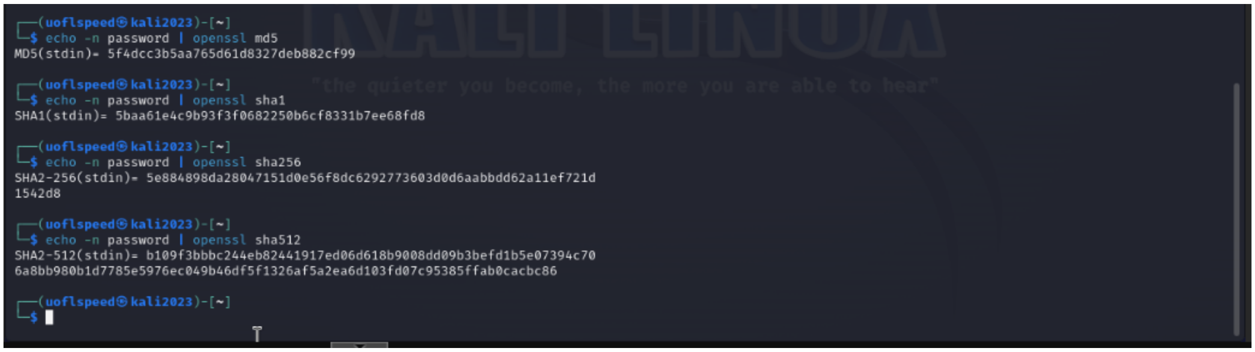
Section 1: Password based authentication

The hash types in this table were found by using Hashcat without a –m specifier, which defaulted Hashcat to attempting to find out the most likely types of hashes for each hash. This was then verified by creating five text files, each of the first four had a separate hash in it, and the fifth had the given password, "password" in it. A dictionary attack was then performed on each hash with the specified –m type to be the top value in the Hashcat assumptions, and when the hash was cracked with the appropriate output, the hash type was then verified and logged in the table.

Hash	Password	Hash Type
5f4dcc3b5aa765d61d8327deb882cf99	password	MD5
5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8	password	SHA1
5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8	password	SHA2-256

b109f3bbbc244eb82441917ed06d618b9008dd09b3befd1b5e07394c706a8bb980b1d7785e5976ec049b46df5f1326af5a2ea6d103fd07c95385ffab0cacbc86	password	SHA2-512

Openssl screenshot:



To crack the four hashes given under a. through d., a Hashcat MD5 dictionary attack using rockyou.txt was initially used. This resulted in hashes 5f4dcc3b5aa765d61d8327deb882cf99, 819b0643d6b89dc9b579fdfc9094f28e, 1c625cc86f824660a320d185916e3c55, and cf7a5b3016903a29bc0c17cfca1e584f being cracked and revealed as password, password3, russia, and may2011, respectively. MD5 was assumed due to hash a. being the same as hash 1. given in the prompt.

The Crypt Linux function was used to hash this password. The structure of the hashed password is separated by $ characters. Below is an image of the hashed passwords in the shadow file.

```
nm-openconnect:!:19648::::::
uoflspeed:$y$j9T$2YrPfd3GgzZnlkONy84nb1$3.s/6SjgOlXKNsdLzQRfBdoiABhLuUUu6c8WIMDUGn0:19648:0:99999:7:::
_galera:!:19648::::::
willow1:$y$j9T$1pHjZzGydgxOCt789D6af1$eDGZttf6PwbWYSA2OMZTCGD.HjRtysp4RLN1GjCkWp6:19884:0:99999:7:::
```

According to the man page for crypt(5), the, y, means that the password is hashed using yescrypt. The second chunk, $j9T$, corresponds to the options used when generating the password hash. The third chunk, $1pHjZzGydgx0Ct789D6af1$, corresponds to the salt used for the hash. The fourth chunk is the hashed password. This information was all gained through the man crypt pages.

Salts are used in hashing to prevent the same two passwords from having the same hash value after being put through the hashing function. The shadow file is used to protect the various passwords of the users on the system. To access the shadow file, sudo permissions have to be given to open the file and read it.

Offline Analysis:

Attempting to crack the root with Hashcat resulted in an error of CL_OUT_OF_RESOURCES. A patch had to be applied to the Windows machine being used for the process. Attempting to crack the root using Hashcat with the rockyou.txt wordlist resulted in Hashcat reporting exhausted. Cracking the user joey with the previously mentioned settings resulted in the password hash being cracked, which showed the password to be jesus. Cracking the user alice resulted in the password hash being cracked, which showed the password to be password1. Cracking the user bob with the previously mentioned settings resulted in Hashcat reporting exhausted as well. Cracking the user tom with the previously mentioned settings resulted in the password hash being cracked, which showed the password to be mookie.

Conclusion

This lab provided hands-on experience with various cryptographic techniques and tools, enhancing our understanding of symmetric encryption, encryption modes, message digests, and the role of Certificate Authorities (CAs). We observed the performance and security differences among encryption algorithms, the importance of choosing appropriate encryption modes like CBC over ECB, and the critical role of hash functions in ensuring data integrity. Additionally, we gained practical knowledge in generating and managing digital certificates and the importance of salting and hashing passwords for security.