# Task 1 – Encoding and Decoding

## Step 1 – Identify Encoding Type

Identify the encoding type (Binary, Decimal, Hexidecimal, Base64) for each of the following strings:

WW91IGhhY2tlciB5b3UhIQ==

- **Base64. It ends with '-' with a mix of letters, numbers, '+', '/'.**

69 110 99 111 100 105 110 103 32 105 115 32 110 111 116 32 101 110 99 114 121 112 116 105 111 110 32 58 41

- **Decimal. It's a series of numbers ranging from 0 to 255**

77 30 30 74 20 77 30 30 74

- **Hexadecimal?**

48 65 78 20 69 73 20 63 6f 6d 6d 6f 6e 6c 79 20 75 73 65 64 20 77 69 74 68 20 61 73 73 65 6d 62 6c 79

- **Hexadecimal. Has Letters A-F.**

01101111 01101110 01100101 00100111 01110011 00100000 01100001 01101110 01100100 00100000 01111010 01100101 01110010 01101111 00100111 01110011

- **Binary. Has only 0s and 1s.**

## Step 2 – Decode

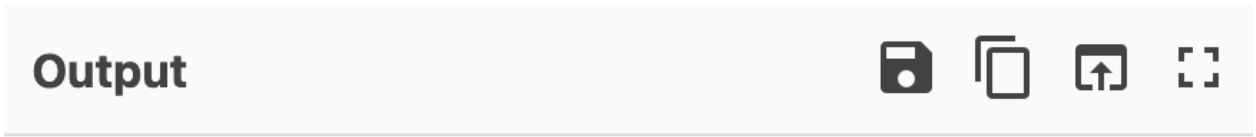
Using Cyber Chef, https://gchq.github.io/CyberChef/, decode each of the strings from step 1.

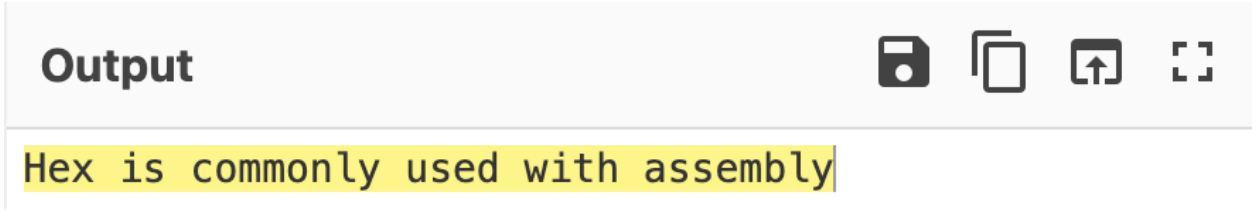
**Output**

```
You hacker you!!
```

**Output**

```
Encoding is not encryption :)
```

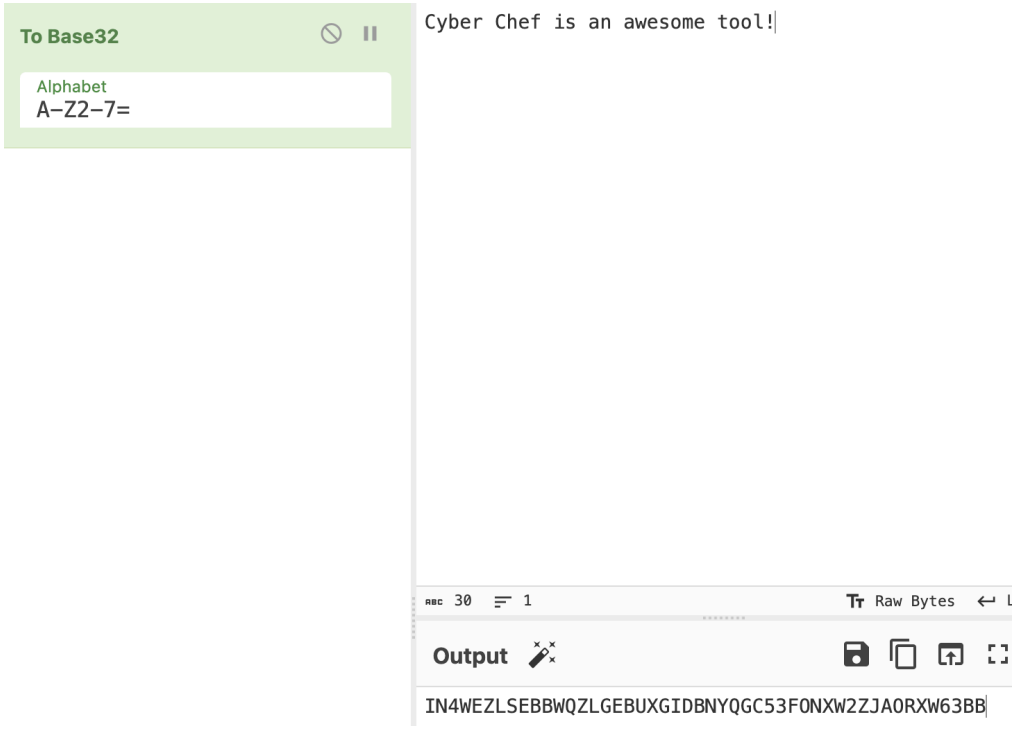



## Step 3 – Encode

Using Cyber Chef, base32 the following string:

Cyber Chef is an awesome tool!



# Task 2 – Key Space

Consider using your Ubuntu or Kali virtual machine for the following task.

## Step 1 – Create Keys of Varying Length

Open a terminal and generate keys of various lengths.

```
openssl rand -base64 32
```

```
openssl rand -base64 1024
```

```
openssl rand -base64 2048
```

```
% openssl rand -base64 32
XMuYfQbKEQ/49AZRaTGprlrnl95Wk2i6DnO2ijBSrAw=
(base) home@Kevins-MacBook-Pro-8 ~
% openssl rand -base64 1024
5Qdvrbf2SQqGtUL+l46cISzEMaAazKXtqxCemNATJG3gwkzi3mnYlyOdngQDNAnZ
```

```
(base) home@Kevins-MacBook-Pro-8 ~
% openssl rand -base64 2048
Iww6UMdxS8EAY/QoDR9EyHuMOuIr+8LMsuVb+XACUF19lKx3WcP2oA281EMqFIb3
```

## Step 2 – Question

Answer the following question. Why is a longer bit key more secure than a shorter bit key?

- It's like having a longer, more secure password. More length = more possible combinations = harder to crack

# Task 3 – OpenSSL Encryption

Consider using your Ubuntu or Kali virtual machine for the following task.

## Step 1 – Create Plaintext

Open a terminal and create a plaintext file that will be encrypted and decrypted in the next steps.

```
echo "some secret message" > plain.txt
```

```
(base) home@Kevins-MacBook-Pro-8 ~
% echo "mr. snuffleupagus" > plain.txt
```

## Step 2 – Encrypt the Plaintext

With the plaintext file created, encrypt the message using AES 256 encryption code block cipher mode.

```
openssl enc -aes-256-cbc -p -in plain.txt -out plain.txt.enc
```

Show that the message was encrypted using cat.

```
cat plain.txt.enc
```

```
(base) home@Kevins-MacBook-Pro-8 ~
% openssl enc -aes-256-cbc -p -in plain.txt -out plain.txt.enc
enter aes-256-cbc encryption password:
Verifying - enter aes-256-cbc encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
salt=CDE26F5F78CBC060
key=F97B5D17F56179CAD3CA757CFBCC89063CD708097BAF4DBE830A791115232123
iv =C4BE7C94A44C9C2FE0BD0F364910C4BE
```

```
(base) home@Kevins-MacBook-Pro-8 ~
% cat plain.txt.enc
Salted__??o_x??`N!&P?'wLʷ?"??{C2.n%
```

## Step 3 – Decrypted the Ciphertext

Decrypt the ciphertext using openssl and the key/password created during encryption.

```
openssl enc -aes-256-cbc -d -A -in plain.txt.enc
```

```
(base) home@Kevins-MacBook-Pro-8 ~
% openssl enc -aes-256-cbc -d -A -in plain.txt.enc
enter aes-256-cbc decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
mr. snuffleupagus
```

# Task 4 – Hash Generation

Consider using your Ubuntu or Kali virtual machine for the following task.

## Step 1 – Create File

Open a terminal and create a file using the following command.

```
echo 'Tamperproof Message: crypto is the coolest!' > message.txt
```

```
(base) home@Kevins-MacBook-Pro-8 ~
% echo 'Tamperproof Message: crypto is the coolest!' > message.txt
(base) home@Kevins-MacBook-Pro-8 ~
```

## Step 2 – Check the MD5 Hash

With the message file created, generate its MD5 hash.

```
md5sum message.txt
```

md5sum command not found, used md5 instead:

```
(base) home@Kevins-MacBook-Pro-8 ~
% md5sum message.txt
zsh: command not found: md5sum
(base) home@Kevins-MacBook-Pro-8 ~
% md5 message.txt

MD5 (message.txt) = 586275d781bbf077bc9b2def6848de9b
```

# Task 5 – Detached Digital Signature

Consider using your Ubuntu or Kali virtual machine for the following task.

## Step 1 – Generate Public Key

Generate a GnuGP public key.

`gpg --gen-key`

```
Change (N)ame, (E)mail, or (O)kay/(Q)uit? o
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: revocation certificate stored as '/Users/home/.gnupg/openpgp-revocs.d/72ED2
862EC3A5CC105F3D3F910A41B3DBD8F4869.rev'
public and secret key created and signed.

pub   ed25519 2024-02-10 [SC] [expires: 2027-02-09]
      72ED2862EC3A5CC105F3D3F910A41B3DBD8F4869
uid                      kevin <kevincendana@csus.edu>
sub   cv25519 2024-02-10 [E] [expires: 2027-02-09]

(base) home@Kevins-MacBook-Pro-8 ~
```

## Step 2 – Export Key

Export the key to your keyring, make sure to use your student email address.

`gpg --armor --output key.gpg --export <YOUR@EMIAL.COM>`

```
(base) home@Kevins-MacBook-Pro-8 ~
% gpg --armor --output key.gpg --export kevincendana@csus.edu
(base) home@Kevins-MacBook-Pro-8 ~
```

## Step 3 – Create Message

Create a message (file) that will be digitally signed.

```
echo "Message integrity and authentication are very cool." > message.txt
```

## Step 4 – Sign Message

Sign the message (file) created in the last step using gpg.

```
gpg --output message.txt.sig --armor --detach-sig message.txt
```

Display and review the signature.

```
cat message.txt.sig
```

```
(base) home@Kevins-MacBook-Pro-8 ~
% echo "Message integrity and authentication are very cool." > message.txt
(base) home@Kevins-MacBook-Pro-8 ~
```

```
(base) home@Kevins-MacBook-Pro-8 ~
% gpg --output message.txt.sig --armor --detach-sig message.txt
File 'message.txt.sig' exists. Overwrite? (y/N) y
(base) home@Kevins-MacBook-Pro-8 ~
% cat message.txt.sig
-----BEGIN PGP SIGNATURE-----

iHUEABYKAB0WIQRVab9VDZfJxIlPPH3mGz/+S1SRKAUCZccZsQAKCRDmGz/+S1SR
KBW+AQC1hxxD0LEAWOOKFRGU7t6+wY0ZmvnEa+NaLH5/tgpy2AD/fJOxWd/rdk8+
T8US+m61gSuJICcosJmeVEmVhtRpagU=
=5Q2X
-----END PGP SIGNATURE-----
(base) home@Kevins-MacBook-Pro-8 ~
```

## Step 5 – Verify Message

Verify the message's integrity and authentication using gpg.

```
gpg --verify message.txt.sig message.txt
```

```
(base) home@Kevins-MacBook-Pro-8 ~
% gpg --verify message.txt.sig message.txt
gpg: Signature made Fri Feb  9 22:37:37 2024 PST
gpg:                using EDDSA key 5569BF550D97C9C4894F3C7DE61B3FFE4B549128
gpg: checking the trustdb
gpg: marginals needed: 3  completes needed: 1  trust model: pgp
gpg: depth: 0  valid:   2  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 2u
gpg: next trustdb check due at 2027-02-09
gpg: Good signature from "kevin <kevincendana@outlook.com>" [ultimate]
(base) home@Kevins-MacBook-Pro-8 ~
```

## Step 6 – Manipulate Message and Reverify

Modify the message and run a gpg verification check to prove that it was tampered with.

```
(base) home@Kevins-MacBook-Pro-8 ~
% echo "changing the message hehehe" > message.txt

(base) home@Kevins-MacBook-Pro-8 ~
```

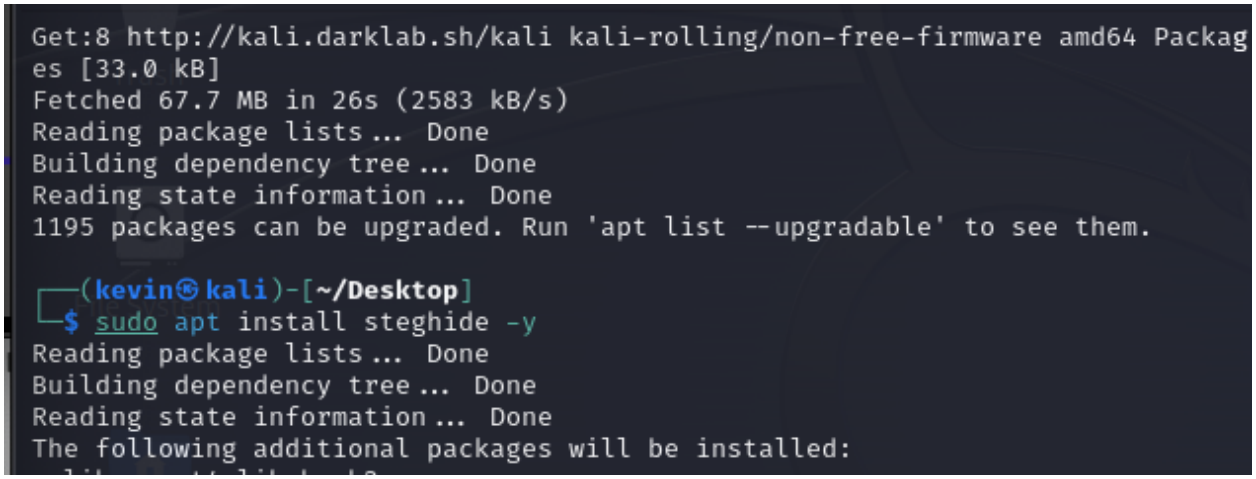
```
(base) home@Kevins-MacBook-Pro-8 ~
% gpg --verify message.txt.sig message.txt

gpg: Signature made Fri Feb  9 22:37:37 2024 PST
gpg:                using EDDSA key 5569BF550D97C9C4894F3C7DE61B3FFE4B549128
gpg: BAD signature from "kevin <kevincendana@outlook.com>" [ultimate]
(base) home@Kevins-MacBook-Pro-8 ~
```

# Task 6 – Steghide

Consider using your Kali virtual machine for the following task.

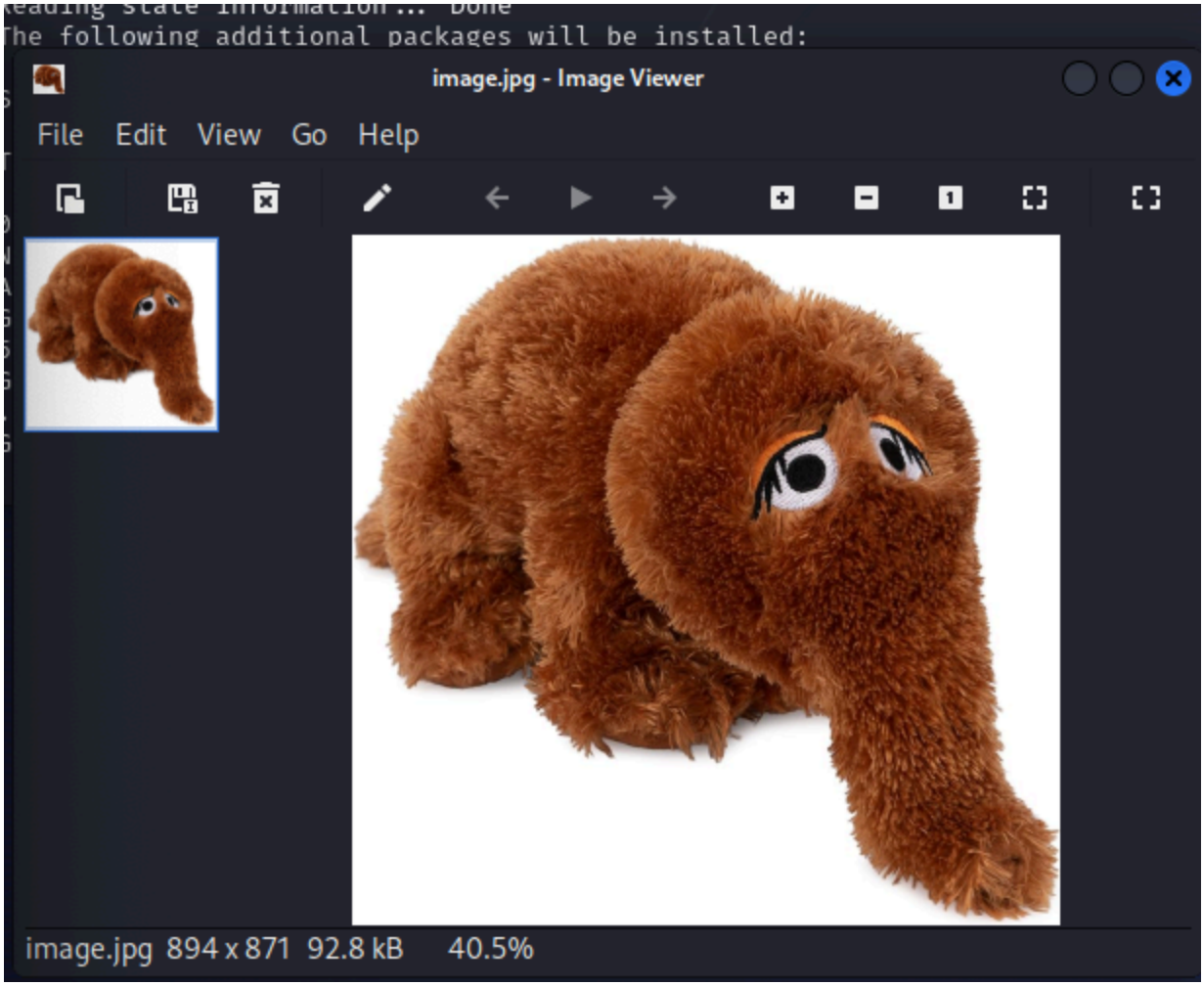
## Step 1 – Install Steghide

Install steghide by updating your system and then downloading and installing the package.




## Step 2 – Obtain a JPG

Download a JPG file from the internet, remember to keep it professional! Consider naming it "image.jpg" and ensure it is a JPG file.




# Step 3 – Create a Secret Message

Create a secret message to hide in the JPG image.

```
echo "Launch Codes: 123123" > secret.txt
```


## Step 4 – Hide the Message

Hide the secret message created in the previous within the JPG image from step 2. Remember your image may be named something other than “image.jpg” used in the following command.

```
steghide embed -ef secret.txt -cf image.jpg
```

```
┌──(kevin㉿kali)-[~/Desktop]
└─$ steghide embed -ef secret.txt -cf image.jpg
Enter passphrase:
Re-Enter passphrase:
embedding "secret.txt" in "image.jpg" ... done

┌──(kevin㉿kali)-[~/Desktop]
└─$
```

## Step 5 – Extract

Extract the secret file from the image.

```
steghide extract -sf image.jpg
```

```
┌──(kevin㉿kali)-[~/Desktop]
└─$ steghide extract -sf image.jpg
Enter passphrase:
the file "secret.txt" does already exist. overwrite ? (y/n) y
wrote extracted data to "secret.txt".

┌──(kevin㉿kali)-[~/Desktop]
```

## Task 7 – Known Plaintext Attack

Consider the following plaintext:

Break my simple encryption

Consider the following ciphertext of the previous plaintext:

rOe nx zlzfvrcya rlpevcgba

*Hint: The ciphertext is composed of alpha characters plus spaces in almost aligned spots to the plaintext.*

### Step 1 – Break the Encryption

Break the encryption and describe the custom algorithm used and its key.

It is okay if you can't break the encryption, but you MUST describe everything you tried that did not work.

Observations:

- **Length**: The output seems to be the same length as the input.
- **Whitespace**: The whitespaces are (almost?) aligned, so maybe most characters are substituted with a single character.
    - It appears as if some characters are taken off of words and added to others. Ex) 'Break' loses 2 characters going to 'rOe' and likely has those 2 characters concatenated to another word, since the overall length of output and input are the same.
- **Mapping**: The mapping doesn't seem consistent at first glance (ex. no A->B, B->C, etc)
    - Characters are not simply scrambled - input has no 'O' while output does.
- **Capitalization**: There is one capital letter for both input and output. That suggests capitalization is never changed, and also suggests that 'B' maps to 'O'.
    - Assuming 'B' -> 'O', we know that the input's words are split apart with white space, because 'Break' is 5 characters and 'rOe' is not.
- **Frequency**: Input has two 'e' and 'm'. Output has 2 'z' and 'l'. From this, we could guess that the encryption is consistent across the same characters - for example, any 'e' might always be a 'z' regardless of other factors.
- **Characters:** Adding up the position of each character for both input and output (ex. 'a' is 1, 'b' is 2) yields an average position of 12.52 vs. 13.09. So the formula seems somewhat consistent, but not always.
- **Punctuation:** There doesn't seem to be any for both input and output.

… I still have no idea what it is. I'm not experienced or knowledgeable enough to string these observations together to form a structured approach to this problem. I give up, but it was fun!