Winter Of Code 2025-26

Information Security

Problem Statement: End-to-End Encrypted Web-Based File Storage and Sharing Platform

Ishan Tiwari
25JE0790

This report will deal with the entire project, starting with some basic concepts and algorithms and my understanding of it. This will be followed by a detailed explanation of the implementations and the code, elaborating every algorithm and function used with emphasis on the key processes and comparisons as to where I decided to

deviate from the norm. I have decided not to focus heavily on website development or UI/UX in this report. This will end with a small demonstration

The project in a nutshell –

I have included the following features over the time span of me working on this project:

1. 2FA based on SHA-1 connected to google authenticate which generates a TOTP (time-based OTP) which every 30 seconds and must be scanned via a QR code.

2. Role-based access control for shared folders and files. These roles will be controlled by the owner/creator of the file only. There are 3 roles that can be assigned to the users with whom the files are shared with, viewer (decrypt and download only, default option), editor (download and edit, such that everyone with a copy of that file will see the changes next time they download it) and revoke (in case a file is shared by mistake, this option will remove the file from the user's terminal).

3. Cloud storage and blind server – The system follows asymmetric cryptography with client-side encryption, and the files are stored in the cloud (Supabase for this one) in encrypted format, such that the server can't access the contents of the file. [Downloading them will result in the ciphertext]

4. Basic utilities like uploading, downloading, updates and sharing are taken care of. The sharing happens through the username (callsign) one registers themselves with.

5. Detailed audit logs with every change related to file (like upload, download, sharing, change of roles or updating data) are recorded and available to view for the owner of that specific file.

Algorithms and Cryptography- What I learned.

AES (Advanced Encryption Standard)

1. **Sources**:

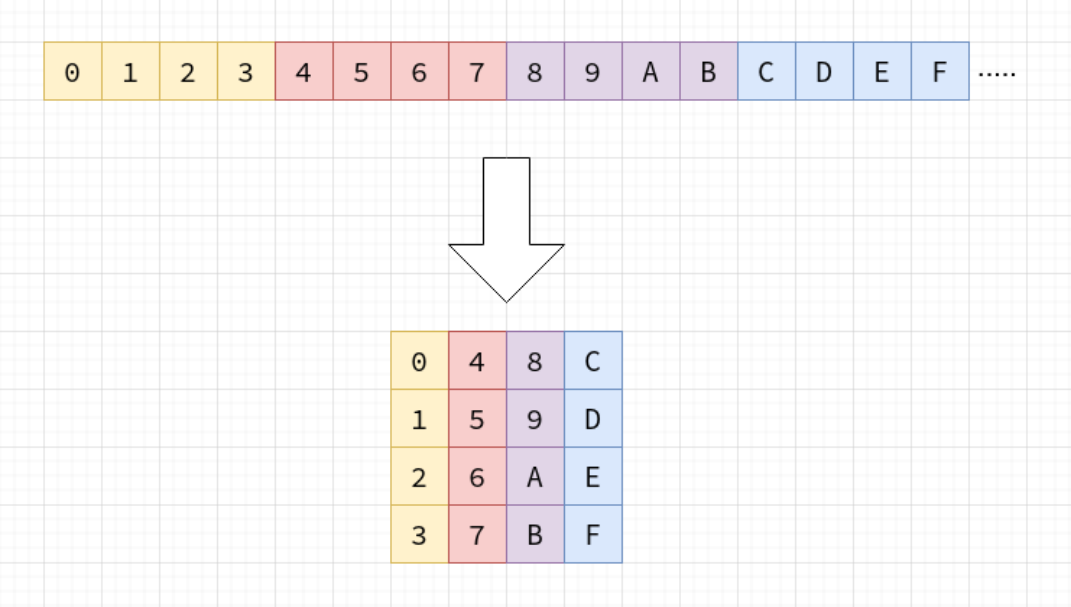
- Francis Strokes' open-source implementation:

https://github.com/francisrstokes/githublog/blob/main/2022/6/15/rolling-your-own-crypto-aes.md

- RFC 3826:

https://datatracker.ietf.org/doc/html/rfc3826

2. **Overview and Algorithm**: AES is a symmetric block cipher (both encryption and decryption are done using the same key). A block cipher operates on multiple bytes (16 in this case) of plaintext at the same time, arranged into a 2D block (in column major order in this case).



The concrete steps of the algorithm are as follows for every block of the plaintext input:

- Key schedule

  The key schedule process is used to take the secret key, and derive an extended set of *round keys*

- Addition of the first-round key

  The first-round key (which is the secret key itself) is "added" to the input. This is not regular addition, but rather addition defined for the finite field *GF(2^8)*, which we will expand on in detail

- A series of "rounds", the exact number of which is defined by the length of the secret key. We will talk about 128-bit keys in this article, in which there are 10 rounds. In a round, a series of operations take place:

  i. Substitute Bytes

  ii. Shift Rows

  iii. Mix Columns

iv. Adding the Round Key

- The final round

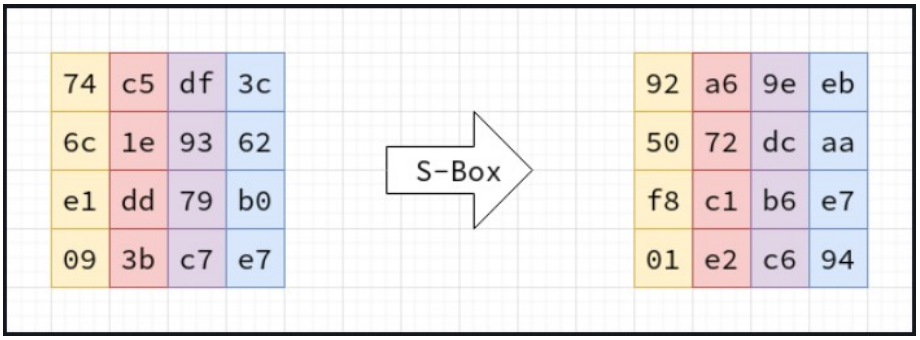
The final round is the same as the previous rounds, except that the *Mix Columns* step is skipped.

3. **Operations and transformations**:

Operations involved in these steps include:

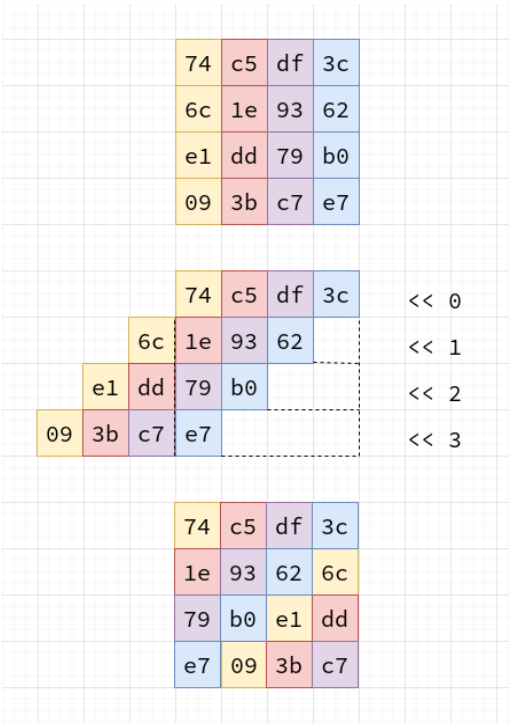
- **Substitute Bytes (SubBytes)**: Takes place during each round and the key scheduling process. In this step, every byte in the block is swapped with a corresponding byte in a lookup table called an *s-box*. The s-box used during encryption is:

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16



A similar kind of inverse s-box is used during decryption.

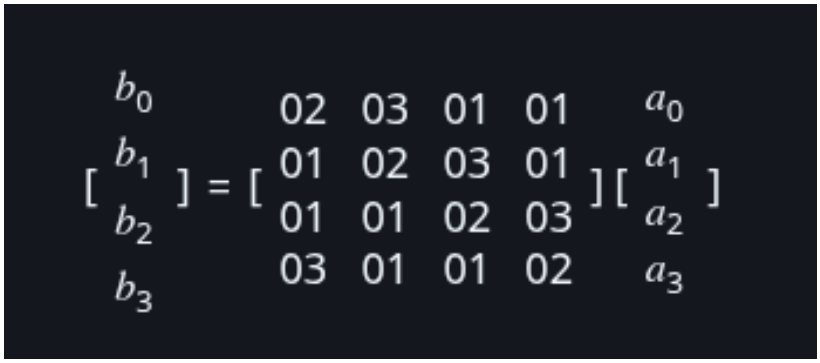
- **Shift Rows**:
  Some of the rows of the current block are shifted. The first row is not shifted at all (alternatively you could think of it having a shift of 0 bytes). The second row is shifted left by 1 byte, the third row 2 bytes, and the fourth row 3 bytes. The "shift" is better described as a *rotation*, because the bytes that are shifted outside of the row come back around into the empty spaces.



- **Mix Columns:**

Provides the property of diffusion (small change in plaintext causes drastic change in ciphertext) in our algorithm.

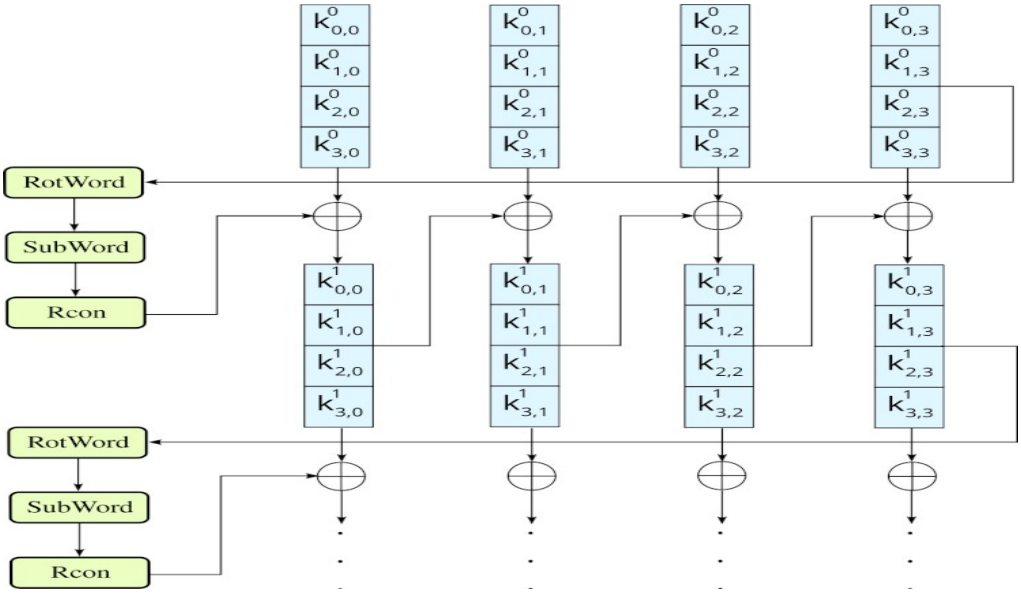
Unlike ShiftRows, this is not simply a transposition of the bytes. Instead, each byte in the column is transformed by combining every byte in the column in a unique but reversible way. Mathematically it is best understood as a vector-matrix multiplication, where the column is the vector, and the matrix is a set of coefficients. The multiplication and addition of each of the elements takes place of course in *GF(2^8)*.



The entire MixColumns operation does this for all 4 columns in the block.

- **Key expansion and the key schedule:**
  AES includes a mechanism for taking the secret key (generally 128-bits in size) and expanding it into a series of

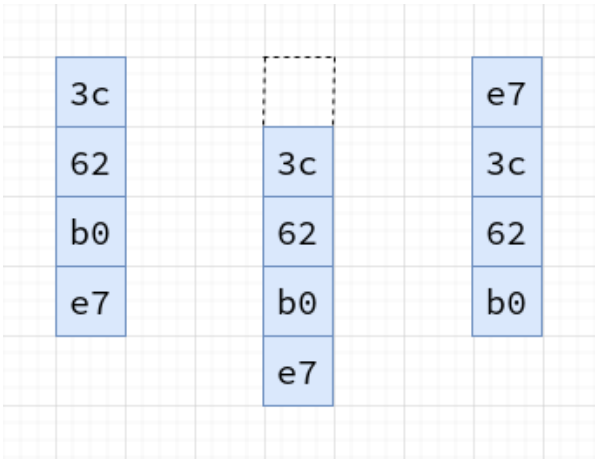
round keys called the *key schedule*.



We start by splitting the secret key into 4 columns. The secret key already constitutes the first-round key. The last column is transformed by 3 operations:

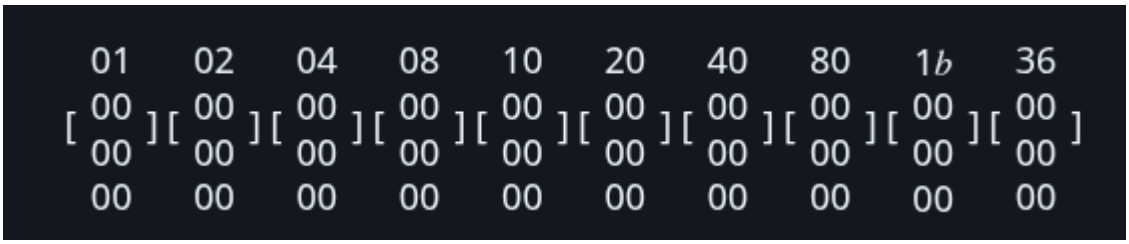
- RotWord
- SubWord
- Rcon

In the RotWord operation, the bytes in the column are *rotated*, much the same as they are in ShiftRows, except that the bytes are only ever rotated by one.



In the SubWord operation, every byte is substituted using the S-box we looked at in the SubBytes step.

Finally, in the Rcon operation (round constant), the column is added to a predefined, constant column corresponding to the current round. The addition here is the xor operation. For a 128-bit key, these constant column vectors are:



After the last column in the initial round key has been transformed by these operations, the *next* 4-column round key is derived using the following steps:

- (Column 1) Adding the transformed column to the first column of the previous round key
- (Column 2) Adding the new round key *Column 1* to the second column of the previous round key
- (Column 3) Adding the new round key *Column 2* to the third column of the previous round key
- (Column 4) Adding the new round key *Column 3* to the fourth column of the previous round key
  This whole process is repeated 10 times (for a 128-byte block), yielding 11 total keys.

- **Add Round Key:** The input block is added (xor-ed) to the corresponding round key, generated during *key expansion*.

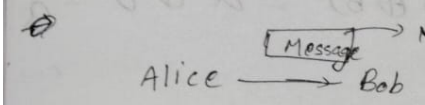
NOTE: All the additions and multiplication operations happen in context of the finite field GF(2^8).

Basics of Cryptography:

The following few pages will contain notes of everything I learnt about fundamentals of cryptography (mostly through pwn.college/wikipedia).

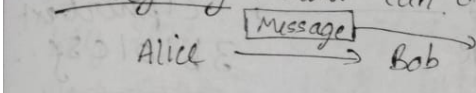
Cryptography

→ **Confidentiality**: Data is not accessible to to users not intentioned for.



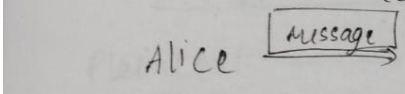
No one reads it along the way.

→ **Integrity**: Data can't be messed with.



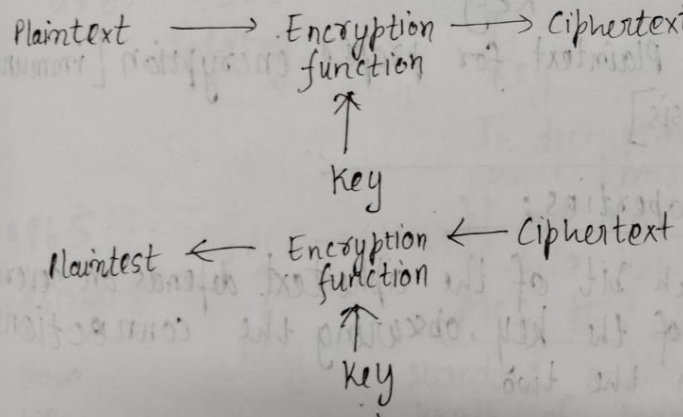
Data is not changed.

→ **Authenticity**: Data is confirmed to be from the correct source.



Bob knows for sure Alice is sending it.

Symmetric Encryption



* **OTP (One-time Pad)**

Encryption:

- Randomly generate a key (equal probability each bit is 0 or 1)
- XOR each bit of plaintext with the key (equal probability each bit is flipped).

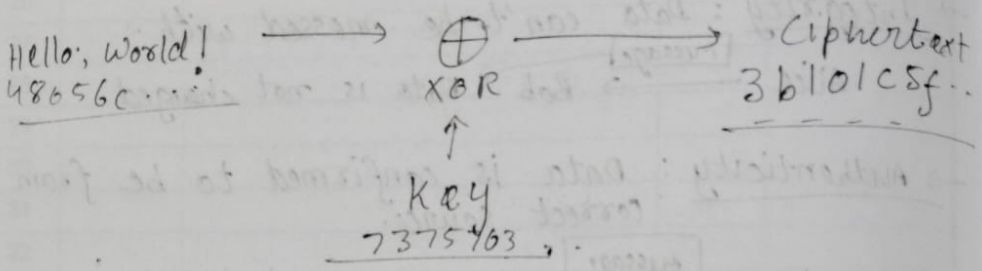
Decryption

↳ XOR each bit of ciphertext with the key.

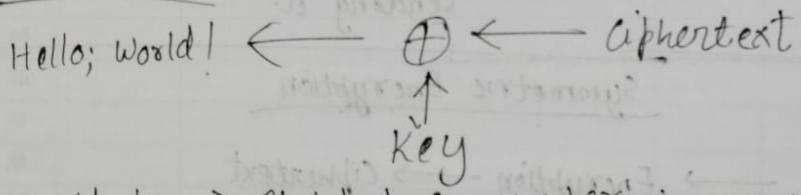
(XOR is its own inverse)

$(a \oplus b) \oplus b = a \oplus (b \oplus b) = a \oplus 0 = a$

Encryption



Decryption



Key should be ≥ Plaintext for proper encryption [immune to frequency analysis]

Encryption Properties:

↳ Confusion: Each bit of the ciphertext depends on several parts of the key, obscuring the connections between the two

a.k.a. change the key slightly to cause avalanche effect in the ciphertext.

↳ Diffusion: A change in a single bit of the plaintext produces a change in about half the bits of ciphertext.

AES [Advanced Encryption Standard]

↳ Achieves these properties

Plaintext ⟶ AES ⟶ ciphertext
↑ Key
& vice-versa for decryption

It is a substitution-Permutation Network.
Key Size : (128/192/256) -bits
Block size : 128-bits [16 bytes]

Encryption Block sizes

Plaintext ⟶ AES ⟶ ciphertext ✗
48656..
--> 13 bytes
↑ Key

U865....000 ⟶ AES ⟶ ciphertext ✓
Padding
↑ Key

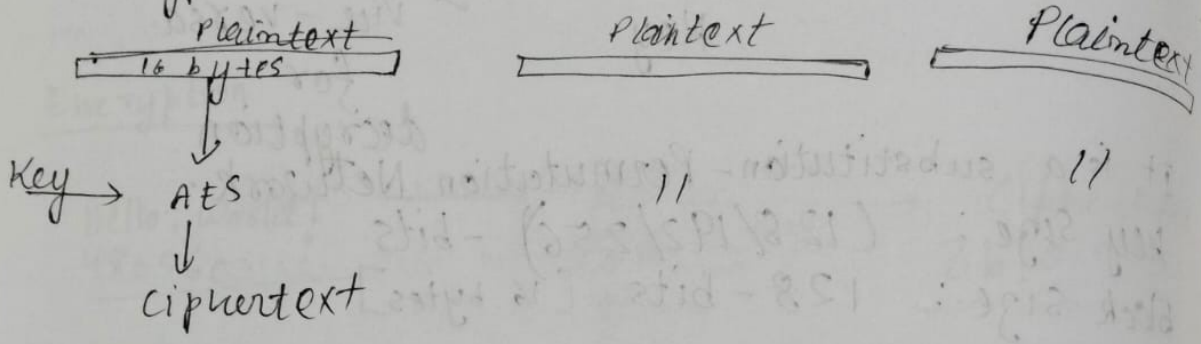
In decryption though, we won't know if null bytes at end are padded or part of data, hence we use PKC #7 here.
standard explicitly mentioning length of padding in the end

9865... ⟶ AES ⟶ ciphertext
...030303
↑ Key

Now we know it is 3 bytes of padding in decryption to get original plaintext

Electronic Codebook [ECB]

Plaintext is divided into 16 byte chunks for encryption.



Decryption works in a similar way with ciphertext being divided into 16 byte chunks.

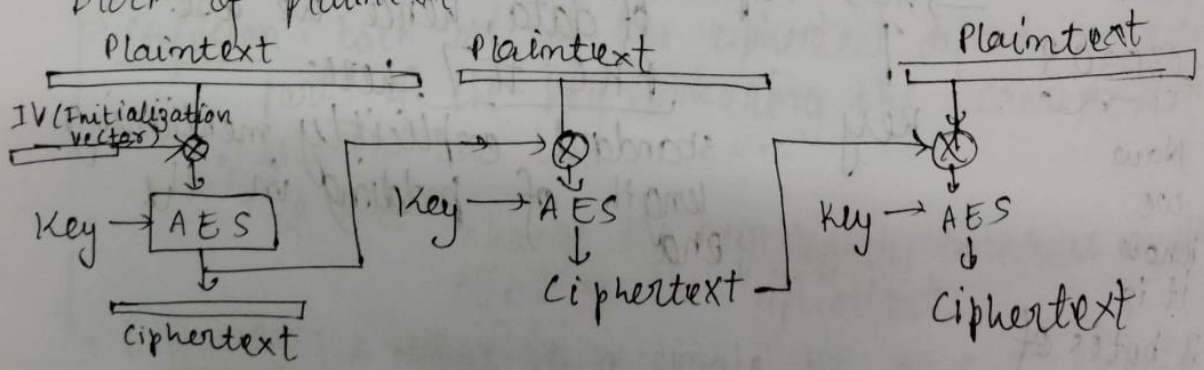
Problem: (Penguin Image)

Repeated data all over the place is shown in the ciphertext too.

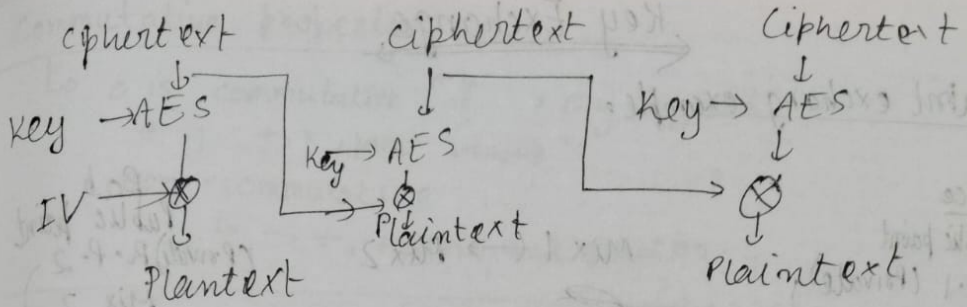
Solution to it:

Cipher Block Chaining: (CBC)

The outputs are chained together, the ciphertext of the previous block is XORed with the next block of plaintext

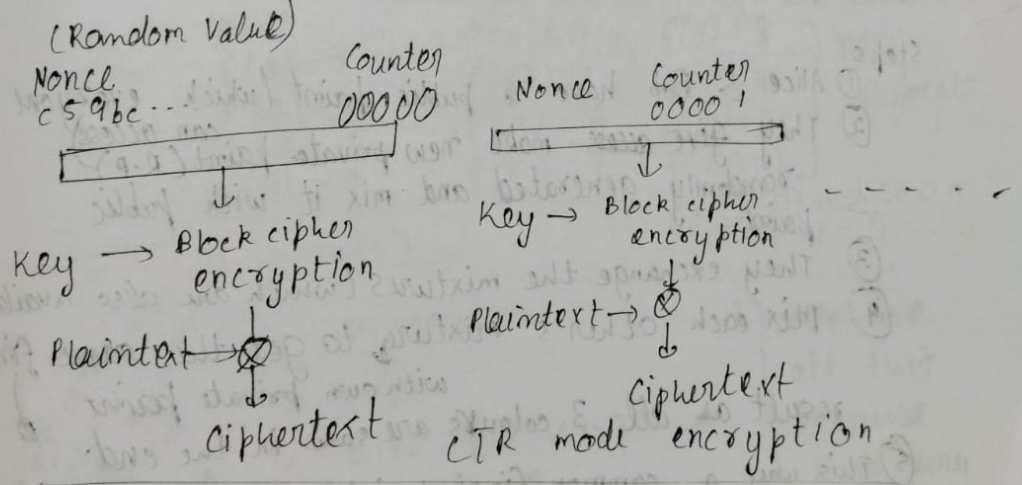


CBC mode encryption

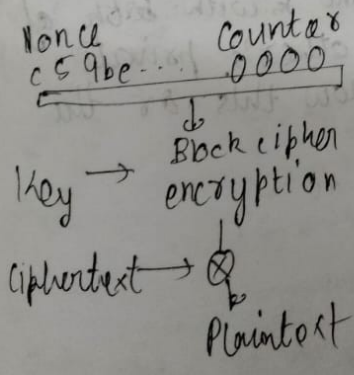


CBC is slower in performance but better in security

Counter (CTR) → Block cipher converted to a stream cipher



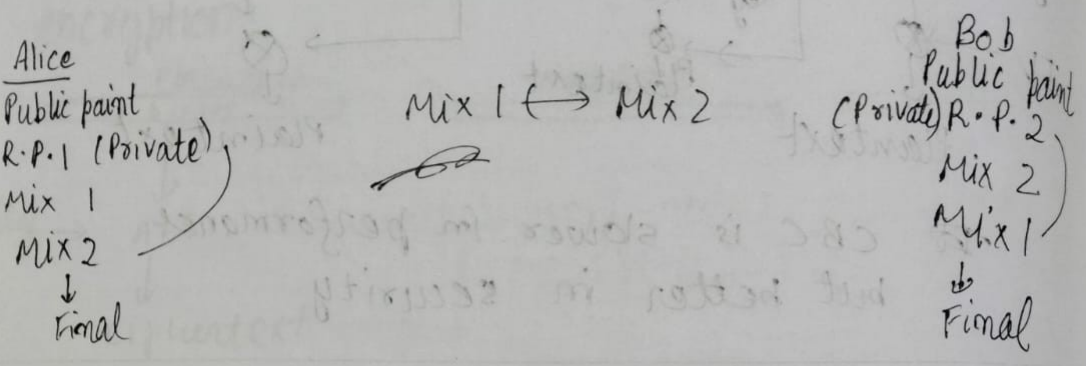
CTR mode encryption

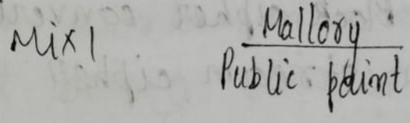


CTR mode decryption faster

Key Exchange

Paint exchang example:





Steps:

1. Alice & Bob have a public paint (which everyone can access)
2. They ~~give access~~ make new private paint (R·P) randomly generated and mix it with public paint.
3. They exchange the mixtures (which are also available)
4. Mix each other's mixture with own private paint to get the same final result as all 3 colours are same in the end.
5. This way a common final paint is with both of them without knowing each other's private paint. (Also others don't know this or the final paint)

Commutative property

↳ $\circ$ is commutative if $x \circ y = y \circ x$

e.g. $+, \times$, paint mixing

non-commutative

↳ $-, \div$, Matrix multiplication.

Modulus Arithmetics

$N \% 12 \rightarrow$ Get number (remainder) of $N \div 12$

⇒ For a prime modulus (generally): [Base = 7 here]

e.g. $7^N \% 13$ ↳ $f(N)$

we get repeating blocks of $13 - 1 = 12$ numbers in seemingly random order (if we don't know we used $7^n \% 13$)

$f(0) = 1$, $f(1) = 7$, $f(2) = 10$, $f(3) = 5$, $f(4) = 9$, $f(5) = 11$, $f(6) = 12$, $f(7) = 6$, $f(8) = 3$, $f(9) = 8$, $f(10) = 4$, $f(11) = 2$, $f(12) = 1$ } Repeats

$f(13) = f(0)$ and so on.

$f(13) = f(1) = 7 \ldots$

[cyclicity works]

$\mathbb{Z}_{13}^{\times}$ system

$7^1 \equiv 7$

$7^5 \equiv 11$ $[7^5 \% 13]$

$7^{-2} \equiv 4$ $[7^{(12-2)} \% 13]$

$7^{1+1} \equiv 10$

$7^{2+2} \equiv 9$

$7^{2 \times 3} \equiv 12$

$7^{4 \times 5} \equiv 3$

$7^{-4 \times 2} \equiv 5$ $[7^{(12-8)} \% 13]$

Use $n-1$ for $\mathbb{Z}_n$

No efficient way to solve this (except brute force) as discrete logarithm problem known

Also

We use all this because it is very difficult to revert e.g. we know $7^5 = 11$ but getting 5 from 11 is very hard if a bigger prime number instead of 7 is used.

Only way is to do it manually. $7^1, 7^2$ & so on.

This operation provides a practical computational one-way transfer.

Diffie-Hellman Key exchange

1. Alice and Bob publicly agree to use a modulus p and base g, where p is prime and g is primitive root modulo p.
2. Alice chooses a secret integer (a), then sends Bob $A = g^a \% p$
3. Bob chooses a secret integer (b), then sends Alice $B = g^b \% p$
4. Alice computes $B^a \% p = s$ } Now both share a
5. Bob computes $A^b \% p = s$ } secret s

$$(g^a \% p)^b \% p = g^{ab} \% p = g^{ba} \% p = (g^b \% p)^a \% p$$

∴ A third person can't take A or B to find s easily unless a & b are known.

Alice		Bob
(23, 5) 23 = p, 5 = g		(23, 5)
7 (a)	17 ⟷ 9	10 (b)
$5^7 \% 23 = 17 = A$		$5^{10} \% 23 = 9$
$\Rightarrow 9^7 \% 23 = 4$		$\Rightarrow 17^{10} \% 23 = 4$

Mallory { 17, (23, 5), 9 } doesn't know 4

Asymmetric Encryption

Two seperate keys, private & public.
(private: decrypts; public: only encrypts)

whole public can encrypt the data but only the person intended to recieve the message will be able to decrypt it.

Fermat's Little Theorem

$$a^p \equiv a \pmod{p}$$

where p is prime [& a is not a multiple of p]

e.g. $7^{13} \% 13 \equiv 7 \% 13 \equiv 7$

Now,

$$a^{p-1} \equiv 1 \pmod{p}$$

where p is prime & a does not divide p

$$a^{p-1} \% p = 1$$

$$\Rightarrow 7^{13-1} \% 13 = 7^{12} \% 13 = 1$$

Euler's theorem

$$\Rightarrow a^{(p-1)(q-1)} \equiv 1 \pmod{pq}$$

where p is prime,
q is prime

$$a^{(p-1)(q-1)+1} \equiv a \pmod{pq}$$

$$a^{\gamma} \equiv a \pmod{pq}$$
$$\gamma \equiv 1 \pmod{(p-1)(q-1)}$$

RSA (Rivest - Shimer - Adleman)

$(m^e)^d \equiv m \pmod{n}$

where $n = pq$,
p is prime,
q is prime,
$ed \equiv 1 \pmod{(p-1)(q-1)}$

$\langle e, n \rangle$ is public key
$\langle d, n \rangle$ is private key
m is plaintext
m^e is ciphertext

Prime factorization ~~[why discrete log is hard]~~.

~~91~~ $7 \times 13 = 91$
$187 = 11 \times 17$ } small numbers

$35266798 09 \times 35490269 27 = 125162816 \ldots 6943$

↑ ez. to do
(multiply two prime nos.)

$6706730448557302609 = 257525665 7 \times 260429597$

↑ Very hard to do vice versa

RSA Key Generation

① choose two large prime numbers p and q.
↳ Efficiently found using a primality test.
② compute $n = pq$
③ Compute $\phi(n) = (p-1)(q-1)$
④ choose e such that $\gcd(e, \phi(n)) = 1$
⑤ 0 most common value of e is $0 \times 10001 = 65537$ (a fermat prime 2^n+1 form)
⑥ Find $d = e^{-1} \pmod{\phi(n)}$
↳ Efficiently found using extended Euclidean algorithm

RSA Encryption

$c \equiv m^e \pmod{n}$] Efficient to calculate
where c is ciphertext,
m is plaintext,
e is public key exponent,
n is key modulus

RSA decryption

$m \equiv c^d \pmod{n}$] Efficient to calculate
where m is plaintext,
c is ciphertext,
d is private key exponent,
n is key modulus

RSA Math

$$m \equiv (m^e)^d \pmod{n}$$

$$(m^e)^d = m^{ed} \equiv (m^d)^e \pmod{n}$$

$$(m^d)^e \equiv m \pmod{n}$$

RSA Signing (Proof or Origin)(Authenticity)

$$s \equiv m^d \pmod{n}$$

where s is signature,
m is plaintext,
d is private key exponent,
n is key modulus

Efficiently calculated using modular exponentiation

[since $ed \equiv 1 \pmod{n}$, a message hashed then encrypt using private key (not public) & if anyone with your public key can decrypt and proof the message REALLY came from you, like a digital sign]

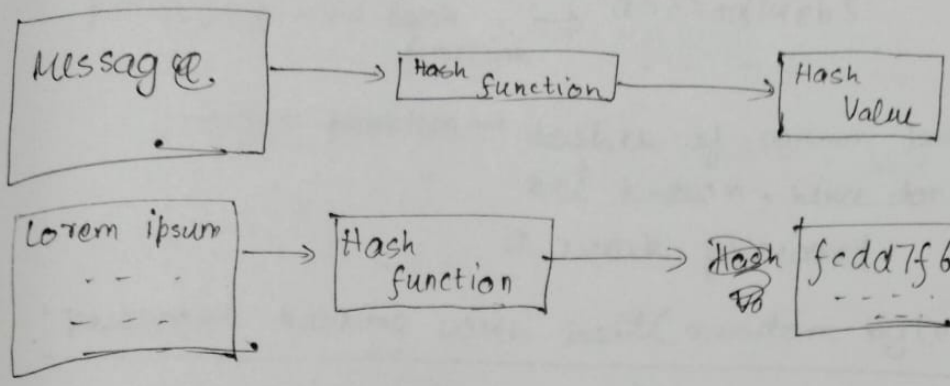
RSA signature Verification

$$m \equiv s^e \pmod{n}$$

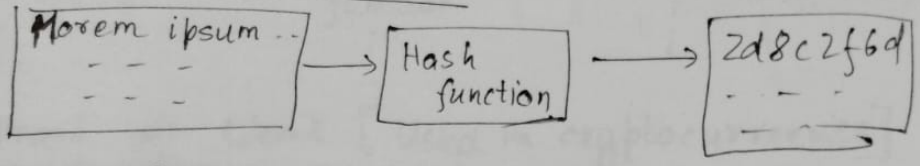
where m is plaintext,
s is signature,
e is public key exponent,
n is key modulus.

Efficiently calculated

Hashing



Avalanche effect



One-Way

Hash value → Hash function → Message ✗

Hash Resistance properties

(1) Pre-image resistance

Given a hash value h, it should be difficult to find any input message 'm' such that

$h = hash(m)$

(2) Second pre-image resistance

Given an input message m1, it should be difficult to find a different input message m2 such that

$hash(m1) = hash(m2)$

(3) Collision resistance: It should be difficult to find two different messages m1 and m2 such that $hash(m1) = hash(m2)$

Password Hashing (before storing the password in the database)

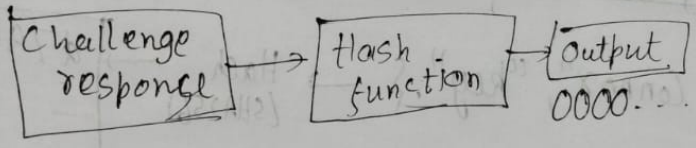
p455w0rd → Hash function → a5c5ad14eb5 . . . . .

slight problem → hashes of common passwords are known, hence don't keep a weak password.

password hashing with salt (random bytes)

p455w0rd + salt → Hash function → d4e0adcae6d . . . .

Proof of Work [Used in cryptocurrencies]



challenge0 → Hash → 05c5 . . .
challenge1 → Hash → d46e . . . .
.
.
.
Challenge 26387 → Hash → 0000 . . .]

Difficulty can be increased by arbitrarily increasing the number of digits in the prefix

Trust

Hashing Certificate Data

{"name": "Connor", "key": {
...
}

→ Hash (SHA256)

d7949ac...

sign RSA

ed3c7fa8...

Certificate Verification

{"name": Connor", "key": { → Hash (SHA256) → d7949ac...
...
}

ed3c7fa8... → Signature verify. RSA → d7949ac...

TOTP generation (using HMAC-SHA-1)

1. **Sources:**

William Henderson's report on Authenticate:

https://whenderson.dev/download/Authenticate__An_Open_Source_TOTP_Authenticator_App.pdf

RFC – 3174, 4226 AND 6238:

https://datatracker.ietf.org/doc/html/rfc3174 (SHA-1)

RFC – 3174, 4226 AND 6238:

https://datatracker.ietf.org/doc/html/rfc3174 (SHA-1)

- First, the plaintext is padded so that it is a multiple of 64 bytes, then it is divided into 64-byte chunks, respectively.
- In each of these chunks, the contents are further divided into 16 integers (4 bytes each). These 16 integers (say W_0 to W_{15}) are used to generate EVEN more numbers up to 80 (W_{16} to W_{79}) using basic bitwise operations defined as:

$$W_t = \mathrm{ROTL}^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})$$

Here ROTL^n is circular left shift of n bits: The bits that fall off the left side come back on the right.

- 5 state variables are initialized as follows:

```
h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0
```

- Now 80 rounds of operations are taken place such that for each round a T is calculated as follows:

$$T = \text{ROTL}^5(H_0) + f_t(H_1, H_2, H_3) + H_4 + W_t + K_t$$

Where H_0 to H_4 are the state variables, W_t denotes the current round integer (from the 80 we generated) and K_t is a round constant (pre-defined from specific values like sqrt2 for round 0 to 19, sqrt3 for 20 to 39, sqrt5 for 40 to 59 and sqrt10 for rounds 60 to 79).
And ft is a function depending on t such that:
for round 0 to 19: $f_t = (H_1 \,\&\, H_2) \mid (\sim H_1 \,\&\, H_3)$
For Round 20 to 39: $f_t = H_1 \oplus H_2 \oplus H_3$
For round 40 to 59: $f_t = (H_1 \,\&\, H_2) \mid (H_1 \,\&\, H_3) \mid (H_2 \,\&\, H_3)$
For round 60 to 79: $f_t = H_1 \oplus H_2 \oplus H_3$

- Then the variables are shuffled in the following manner before the next round:

$$H_4 = H_3$$

$$H_3 = H_2$$

$$H_2 = \text{ROTL}^{30}(H_1)$$

$$H_1 = H_0$$

$$H_0 = T$$

- After 80 such rounds, the state variables (H_0 to H_4) have been thoroughly mixed. They are added (via 2^32 modular addition, aka, if they get too big, they just wrap around 0) to the original state (before the loop started).

$$H_0 = H_0 + H_{0\_initial}$$

$$H_1 = H_1 + H_{1\_initial}$$

$$H_2 = H_2 + H_{2\_initial}$$

$$H_3 = H_3 + H_{3\_initial}$$

$$H_4 = H_4 + H_{4\_initial}$$

This is followed by combining all these into a singular 160-bit hash value.

$$\text{Final Hash} = H_0 \,||\, H_1 \,||\, H_2 \,||\, H_3 \,||\, H_4$$

HMAC (Hash – based Message Authentication Code):

Introduces a secret key(K) into the hash to ensure authenticity via a "keyed hash".

- Define inner padding (ipad), outer padding (opad), normalised Key (K^{l}) as follows:
  ipad: The byte 0x36 repeated 64 times.
  opad: The byte 0x5C repeated 64 times.
  K^{l} is derived from K as follows:
  If K is less than 64 bytes – Padded with zeros on the right.
  If K is more than 64 bytes – Hashed with SHA-1 to get a 20-byte value which is then padded with zeroes until it reaches 64 bytes.

$$K' = \begin{cases} \mathrm{H}(K) & K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

- Inner Hash:
  The normalised key is XORed with ipad. The original message is then appended to this result. This string is made to run through SHA-1.
- Outer Hash:
  The normalised key is XORed with opad. The output of the inner hash is appended to this and then the whole thing is hashed through SHA-1.

$$\mathrm{HMAC}(K, m) = \mathrm{H}\left(\left(K' \oplus opad\right) \parallel \mathrm{H}\left(\left(K' \oplus ipad\right) \parallel m\right)\right)$$

HOTP (HMAC – based one-time password):
Generates the actual 6-digit OTP the user generally enters from the 20-byte hash value through truncation.

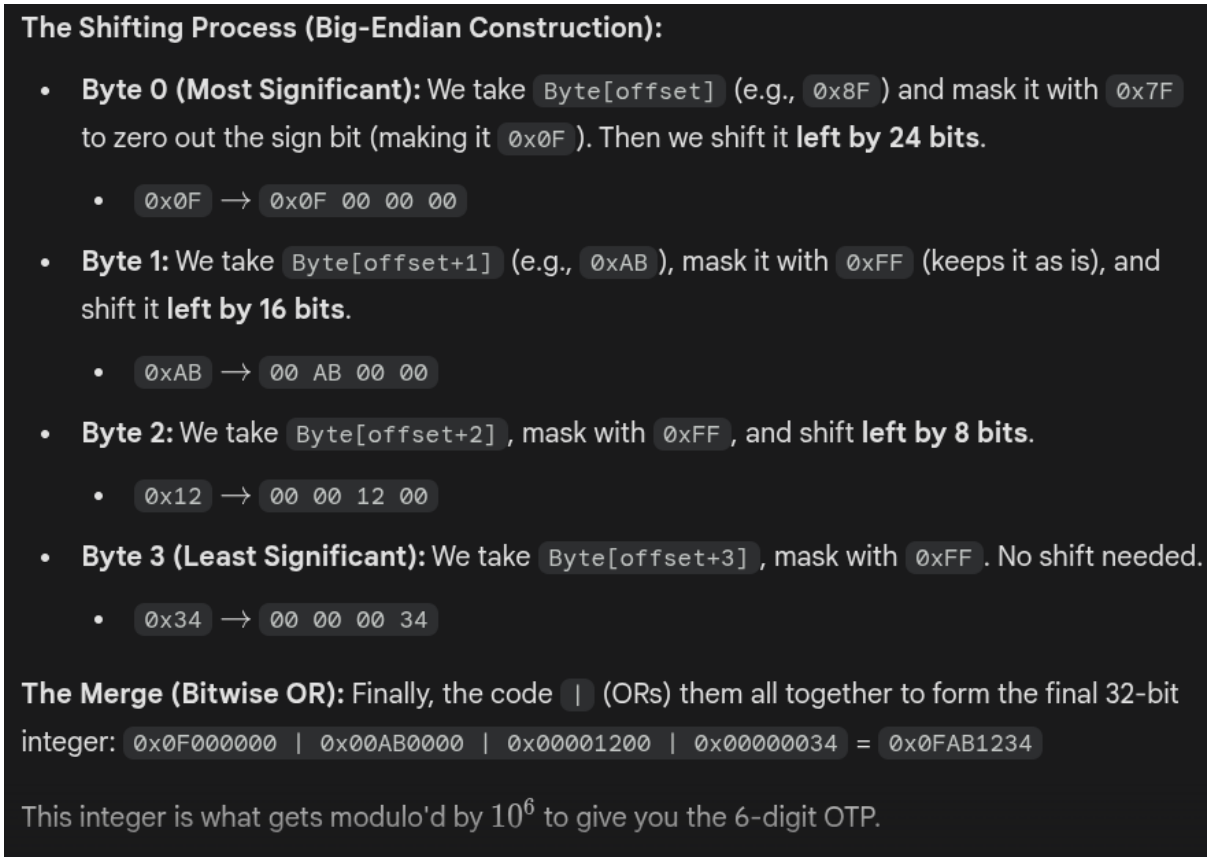
- Define the secret key (K) and the counter (C):
  K: Shared key between client and server;

C: A number initialised to 0 which increments every time a password is generated.
Keeping the counter in sync between client and server is often and issue, hence this code is often not used.

- Generate a 20 – byte code through HMAC first.
  HS = HMAC-SHA1(K, C)
- Initialising the offset: Take the last byte of the HMAC result (byte 19). Extract the lower 4 bits of this byte.
  Offset = HS[19] & 0x0F
- Go to the byte at the Offset position.
  Take that byte and the next three bytes.
  P = HS[Offset] || HS[Offset+1] || HS[Offset+2] || HS[Offset+3]
- To avoid confusion (and negative numbers), we mask the most significant bit.
  Perform a bitwise **AND** on the first byte with 0x7F.
  Binary Code = P & 0x7FFFFFFF
  (This forces the first bit to be 0, ensuring the number is treated as a positive 31-bit integer).

```
// Truncate (Dynamic Offset)
int offset = hmac_result[19] & 0x0F;
uint32_t binary_code =
      ((hmac_result[offset] & 0x7F) << 24)
    | ((hmac_result[offset + 1] & 0xFF) << 16)
    | ((hmac_result[offset + 2] & 0xFF) << 8)
    | (hmac_result[offset + 3] & 0xFF);

// Modulo 10^6
return binary_code % 1000000;
```

Finally, find the modulo by 10^6 to get a 6-digit OTP.
The following explanation by Gemini sums up the Big-Endian Construction accurately.



TOTP:

Same as HOTP, except it uses the current UNIX timestamp as the counter.

Implementation and code – The working explained

This is the main section practically, here we will go over the block cipher used, the code for TOTP generation and the essential working of the site

Block cipher:

This is the file encryption module; it uses a 16-round Feistel Network. It uses Affine transformation inside the Feistel Network. This essentially narrows down to modular arithmetic: a number has a multiplicative inverse modulo 2^{32} if and only if that number is odd.

The core operation inside my feistel_function is a linear congruential generator (LCG) step:

$$T = (x \cdot \text{multiplier}) + \text{constant} \pmod{2^{32}}$$

This corresponds to the affine cipher function $f(x) = ax + b$.

Case 1: MFA Success (The Reversible Path)

When mfa_status == 1, the multiplier a is set to 31337 (odd).

Math: gcd(31337, 2^{32}) = 1.

Result: 31337 has a modular multiplicative inverse. The operation is bijective (1-to-1 mapping), preserving all information (entropy). The decryption proceeds correctly.

Case 2: MFA Failure (The Destructive Path)

When mfa_status == 0, the multiplier a is set to 31336 (even).

Math: gcd(31336, 2^{32}) = 1. They share a factor of 2.

Result: This multiplier is **not invertible**. It collapses the state space—multiple inputs map to the same output (collisions).

The Trap: This operation mathematically destroys information. Even if the user has the correct password (keys), the wrong OTP forces the cipher into a "lossy" mode. The output looks like a valid ciphertext, but it decodes into garbage, effectively "poisoning" the session without alerting the attacker.

Given below is the explanation of the code and what it each function does:

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>


#define NUM_ROUNDS 16

static uint32_t rotl32(uint32_t x, int n) {
    return (x << n) | (x >> (32 - n));
}

uint32_t feistel_function(uint32_t R, uint32_t key, int mfa_status) {
    uint32_t t = R ^ key;

    // MFA Valid (1) -> Multiplier 31337 (Odd/Invertible)
    // MFA Fail  (0) -> Multiplier 31336 (Even/Destructive)
    uint32_t multiplier = (mfa_status == 1) ? 31337 : 31336;

    t = (t * multiplier) + 0x12345678;
    return rotl32(t, 5);
}
```

rotl32: left rotation of x by n bits.

feistel_function: First the input block R (right half) is XORed with the round key. Then based upon whether the OTP (mfa status) is correct or not, selects the respective multiplier.

Then

t = (t * multiplier) + 0x12345678; is the Affine transformation. This implements the function f(x) = ax + b mod{2^{32}}. This is then diffused by rotating left by 5 units.

Note: unsigned integers of specified bit size (uint32_t or uint_8t) are used instead of int because in C a standard int can vary depending on the machine and cryptographic functions require specific size of such things. Also unsigned integers have a wrapping property (modulo 2^{32}) critical for modular arithmetic, signed integers have undefined behaviour on overflow, causing problems.

Moving on now,

```c
void generate_keys(const char *password, uint32_t *keys) {
    uint32_t seed = 0;
    while (*password) {
        seed = ((seed << 5) + seed) + *password++;
    }
    for(int i=0; i<NUM_ROUNDS; i++) {
        keys[i] = seed + (i * 0x9E3779B9);
        seed = rotl32(seed, 3);
    }
}

void feistel_block_operate(uint32_t *left, uint32_t *right, uint32_t *keys, int mfa_status) {
    uint32_t temp;
    for (int i = 0; i < NUM_ROUNDS; i++) {
        uint32_t old_right = *right;
        uint32_t f_out = feistel_function(*right, keys[i], mfa_status);
        *right = *left ^ f_out;
        *left = old_right;
    }
    temp = *left; *left = *right; *right = temp;
}

void print_output_as_text(uint32_t L, uint32_t R) {
    char buf[9];
    memcpy(buf, &L, 4);
    memcpy(buf + 4, &R, 4);
    buf[8] = '\0';

    printf(" -> Text: \"");
    for(int i=0; i<8; i++) {
        unsigned char c = buf[i];
        if (c >= 32 && c <= 126) printf("%c", c);
        else printf("\\x%02X", c);
    }
    printf("\"");
}
```

generate_keys: Key scheduling process, turns a password into useful keys.

A djb2 variant is used for password hashing, the line

seed = ((seed << 5) + seed) + *password++;

Runs through each character in the password and for each character, shifts the seed by left by 5 bits (multiplying by 32) and then add it to itself. This ensures diffusion, such that a small change in password causes vast change in seed.

Then in key expansion, keys[i] = seed + (i * 0x9E3779B9);

0x9E3779B9 is the hexadecimal representation of the Golden Ratio (phi) scaled to 32 bits (2^{32} / phi). Commonly used in cryptography (e.g. TEA cipher), adding its multiples ensures randomness in each of the 16 rounds. After the key generation, the seed itself is rotated by 3 bits ensuring making sure of no relation between the consecutive keys.

feistel_block_operate: This takes the data and runs it through 16 rounds of encryption.

1. The Inputs:

a. *left and *right: The 64-bit data block is split into two 32-bit halves.

b. mfa_status: The flag (0 or 1) that determines if we are in "Destructive" or "Valid" mode.

2. The Loop (16 Rounds):

a. Save State: uint32_t old_right = *right; saves the current Right half because it will become the Left half in the next round.

b. The F-Function: f_out = feistel_function(...) calculates the scrambled value using the current Right half and the current Round Key. This is where the Trapdoor logic executes.

c. XOR Mixing: *right = *left ^ f_out; applies the scrambled data to the Left half. This is the beauty of Feistel networks: we encrypt the Left half using the Right half, without needing the F-function to be invertible itself (though our trapdoor relies on it).

d. The "Swap": *left = old_right; moves the original Right half to the Left position for the next round.

3. The Final Swap:

a. Code: temp = *left; *left = *right; *right = temp;

b. Purpose: In a standard Feistel network, the Left and Right halves are swapped one last time after the final round. This ensures that decryption is the exact reverse of encryption (just using the keys in reverse order).

print_output_as_text:

1. Reconstruction: memcpy is used to copy the two 32-bit integers (L and R) back into a continuous 8-byte character array (buf). This reverses the splitting process done at the start of encryption.

2. Safe Printing:
    a. The loop checks every byte.
    b. Printable: If the byte represents a standard ASCII character (value 32-126, e.g., 'a', 'B', '$'), it prints the character.
    c. Non-Printable: If the byte is a control code or binary garbage (which happens if decryption fails), it prints the Hex code (e.g., \x0F) instead of crashing the terminal or making a beep sound.

```
// 1. ENCRYPT EXPORT


void wasm_encrypt_file(uint8_t* data, int length, char* password) {
    uint32_t keys[NUM_ROUNDS];
    generate_keys(password, keys);

    uint32_t* blocks = (uint32_t*)data;
    int block_count = length / 8;

    for (int i = 0; i < block_count; i++) {
        feistel_block_operate(&blocks[i*2], &blocks[i*2 + 1], keys, 1);
    }
}

// 2. DECRYPT EXPORT

void wasm_decrypt_file(uint8_t* data, int length, char* password, int mode) {
    uint32_t keys[NUM_ROUNDS];
    generate_keys(password, keys);

    uint32_t decrypt_keys[NUM_ROUNDS];
    for(int i=0; i<NUM_ROUNDS; i++) decrypt_keys[i] = keys[NUM_ROUNDS - 1 - i];

    int mfa_status = mode;

    uint32_t* blocks = (uint32_t*)data;
    int block_count = length / 8;

    for (int i = 0; i < block_count; i++) {
        feistel_block_operate(&blocks[i*2], &blocks[i*2 + 1], decrypt_keys, mfa_status);
    }
}
```

wasm_encrypt_file:

- Key Expansion: It first derives a set of sub-keys (keys[NUM_ROUNDS]) from the user's password. This ensures that the security of the encryption is directly tied to the entropy of the user's credentials.
- Type Casting: The raw byte stream (uint8_t*) is cast to 32-bit integer blocks (uint32_t*). This is necessary because the Feistel network operates on two 32-bit halves (Left and Right) to form a 64-bit block.

- The Hardcoded "True" State:

  The critical line is:

  feistel_block_operate(&blocks[i*2], &blocks[i*2 + 1], keys, 1);

  The final parameter 1 hardcodes the mfa_status to Valid. This forces the underlying Feistel function to use the mathematically invertible multiplier (31337), ensuring the data is encrypted correctly and can be recovered later.

wasm_decrypt_file:

New Parameter: mode (Integer) - Received from the server. 1 indicates successful 2FA; 0 indicates a failed/trap state.

Key Reversal:

Standard Feistel decryption requires applying the sub-keys in reverse order. The loop decrypt_keys[i] = keys[NUM_ROUNDS - 1 - i] handles this inversion perfectly.

The Trap Injection:

Instead of hardcoding the status, the function passes the mode variable deep into the cipher core:

feistel_block_operate(&blocks[i*2], &blocks[i*2 + 1], decrypt_keys, mfa_status);

- If mode == 1 (Authenticated): The function uses the invertible multiplier (31337). The math reverses perfectly: $Dec(Enc(M)) = M$.
- If mode == 0 (Trap): The function uses the "destructive" multiplier (31336). This introduces a deliberate mathematical error in every single round of the cipher.

TOTP generation:

```c
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdint.h>
#include <time.h>


#define ROTLEFT(value, bits) (((value) << (bits)) | ((value) >> (32 - (bits))))

void sha1_expand_block(const uint8_t* block_64bytes, uint32_t* W) {

    for (int i = 0; i < 16; i++) {

        W[i] = ((uint32_t)block_64bytes[i * 4]     << 24) |
               ((uint32_t)block_64bytes[i * 4 + 1] << 16) |
               ((uint32_t)block_64bytes[i * 4 + 2] << 8)  |
               ((uint32_t)block_64bytes[i * 4 + 3]);
    }

    for (int i = 16; i < 80; i++) {

        uint32_t temp = W[i - 3] ^ W[i - 8] ^ W[i - 14] ^ W[i - 16];
        W[i] = ROTLEFT(temp, 1);
    }
}
```

ROTLEFT: Left rotation by 32 bits.

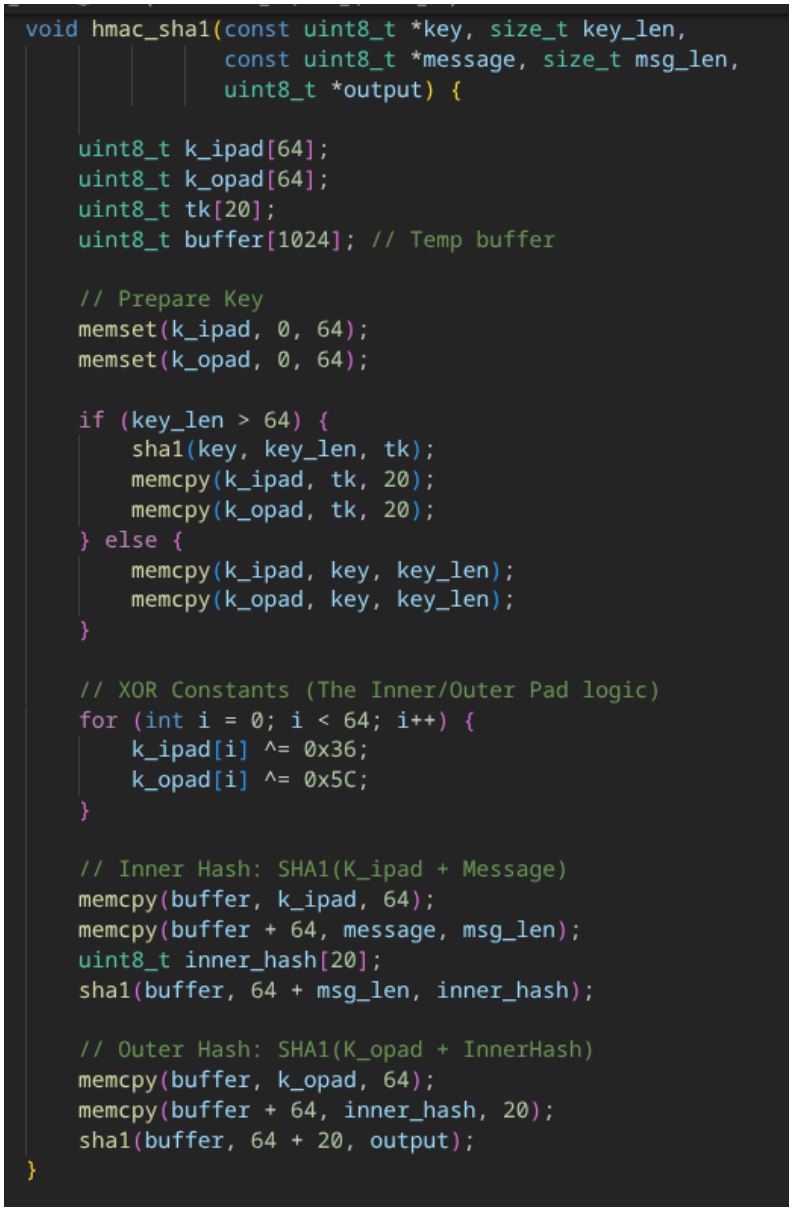
sha1_expand_block: Generates the 80 numbers, W[0] to W[79] from the plaintext blocks and then rotates for diffusion.

```c
void sha1_compress_cycle(uint32_t* W, uint32_t* state) {
    uint32_t a = state[0];
    uint32_t b = state[1];
    uint32_t c = state[2];
    uint32_t d = state[3];
    uint32_t e = state[4];
    uint32_t f, k, temp;
    for (int i = 0; i < 80; i++) {
        if (i <= 19) {
            f = (b & c) | ((~b) & d);
            k = 0x5A827999;
        }
        else if (i <= 39) {
            f = b ^ c ^ d;
            k = 0x6ED9EBA1;
        }
        else if (i <= 59) {
            f = (b & c) | (b & d) | (c & d);
            k = 0x8F1BBCDC;
        }
        else {
            f = b ^ c ^ d;
            k = 0xCA62C1D6;
        }
        temp = ROTLEFT(a, 5) + f + e + k + W[i];

        e = d;
        d = c;
        c = ROTLEFT(b, 30);
        b = a;
        a = temp;
    }
    state[0] += a;
    state[1] += b;
    state[2] += c;
    state[3] += d;
    state[4] += e;
}
```

sha1_compress_cycle: This is the main loop of the sha_1, taking the current hash value (state) into a,b,c,d,e and then running the loop 80 times, using each W[i] in the process. The logic of each of 20 round parts is defined as well and temp is calculated, after that the values are swapped and added back to the main state variable to update the hash.

```c
void sha1(const uint8_t *message, size_t len, uint8_t *digest) {
    uint32_t state[5] = {0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476, 0xC3D2E1F0};
    uint32_t W[80];
    uint8_t buffer[64];

    // Calculate full padded length (Msg + 0x80 + Zeros + 8-byte Length)
    size_t full_len = len + 1 + 8;
    size_t padded_len = (full_len + 63) / 64 * 64;

    for (size_t i = 0; i < padded_len; i += 64) {
        memset(buffer, 0, 64);

        // Copy Message
        if (i < len) {
            size_t bytes_to_copy = (len - i) < 64 ? (len - i) : 64;
            memcpy(buffer, message + i, bytes_to_copy);
        }

        // Add 0x80 Byte (Padding Start)
        if (i <= len && len < i + 64) {
            buffer[len - i] = 0x80;
        }

        // Add Length in Bits (Big Endian) at the end of the last block
        if (i + 64 >= padded_len) {
            uint64_t bit_len = (uint64_t)len * 8;
            for(int j=0; j<8; j++) {
                buffer[63-j] = (bit_len >> (j*8)) & 0xFF;
            }
        }
        sha1_expand_block(buffer, W);
        sha1_compress_cycle(W, state);
    }

    for (int i = 0; i < 5; i++) {
        digest[i*4]     = (state[i] >> 24) & 0xFF;
        digest[i*4 + 1] = (state[i] >> 16) & 0xFF;
        digest[i*4 + 2] = (state[i] >> 8)  & 0xFF;
        digest[i*4 + 3] = state[i] & 0xFF;
    }
}
```

sha1: Starts with defining the 5 states and doing the padding. After the blocks are made and padded, it runs through the message one block (64 bytes) at a time. 0x80 is added at the end of message to act as a delimiter and begin the padding. The final 8 bytes are updated with the message length and are used to set up the big endian. The whole thing is then put through the expansion and compression functions defined earlier. Then all this is stored in the digest array, our 20 – byte hash.



```c
void hmac_sha1(const uint8_t *key, size_t key_len,
               const uint8_t *message, size_t msg_len,
               uint8_t *output) {

    uint8_t k_ipad[64];
    uint8_t k_opad[64];
    uint8_t tk[20];
    uint8_t buffer[1024]; // Temp buffer

    // Prepare Key
    memset(k_ipad, 0, 64);
    memset(k_opad, 0, 64);

    if (key_len > 64) {
        sha1(key, key_len, tk);
        memcpy(k_ipad, tk, 20);
        memcpy(k_opad, tk, 20);
    } else {
        memcpy(k_ipad, key, key_len);
        memcpy(k_opad, key, key_len);
    }

    // XOR Constants (The Inner/Outer Pad logic)
    for (int i = 0; i < 64; i++) {
        k_ipad[i] ^= 0x36;
        k_opad[i] ^= 0x5C;
    }

    // Inner Hash: SHA1(K_ipad + Message)
    memcpy(buffer, k_ipad, 64);
    memcpy(buffer + 64, message, msg_len);
    uint8_t inner_hash[20];
    sha1(buffer, 64 + msg_len, inner_hash);

    // Outer Hash: SHA1(K_opad + InnerHash)
    memcpy(buffer, k_opad, 64);
    memcpy(buffer + 64, inner_hash, 20);
    sha1(buffer, 64 + 20, output);
}
```

hmac_sha1: Implements the HMAC algorithm. First checks the key length, if greater than 64 bytes, it is hashed down to 20. Then the inner padding and outer padding are XORed into the key. Then the inner hash and outer hash are calculated respectively.

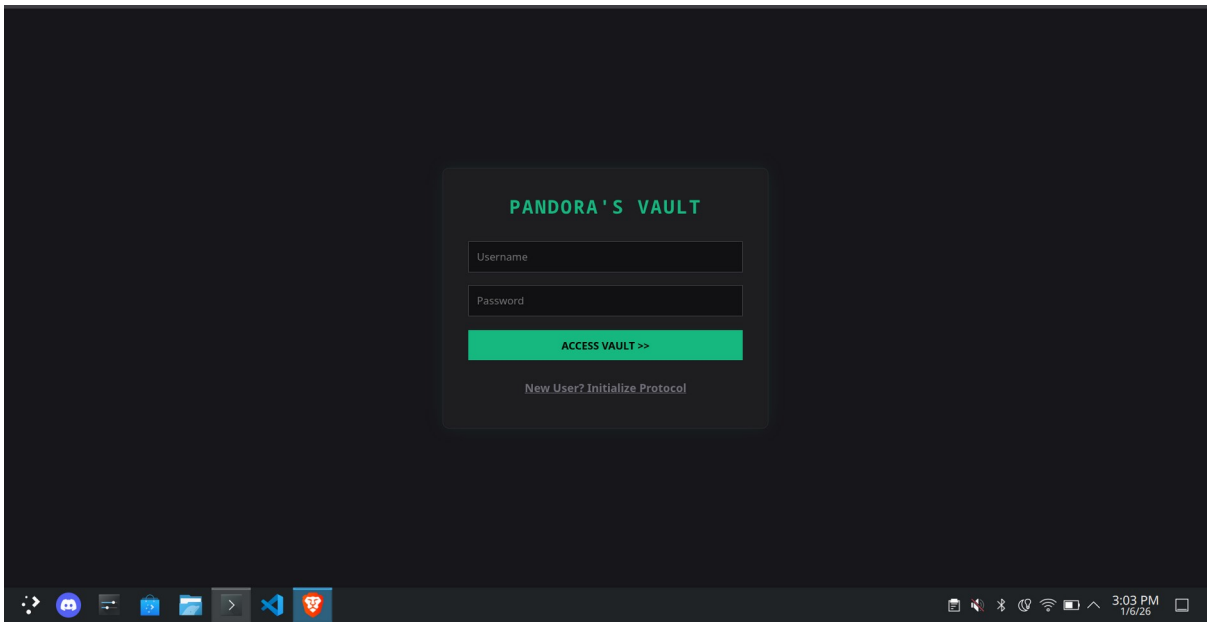
```c
int generate_totp(const uint8_t *secret, size_t secret_len, uint64_t time_step) {
    uint8_t hmac_result[20];

    // Convert Time to Bytes (Big Endian)
    uint8_t counter_bytes[8];
    for(int i=7; i>=0; i--) {
        counter_bytes[i] = time_step & 0xFF;
        time_step >>= 8;
    }

    // Generate HMAC
    hmac_sha1(secret, secret_len, counter_bytes, 8, hmac_result);

    // Truncate (Dynamic Offset)
    int offset = hmac_result[19] & 0x0F;
    uint32_t binary_code =
          ((hmac_result[offset] & 0x7F) << 24)
        | ((hmac_result[offset + 1] & 0xFF) << 16)
        | ((hmac_result[offset + 2] & 0xFF) << 8)
        | (hmac_result[offset + 3] & 0xFF);

    // Modulo 10^6
    return binary_code % 1000000;
}
```

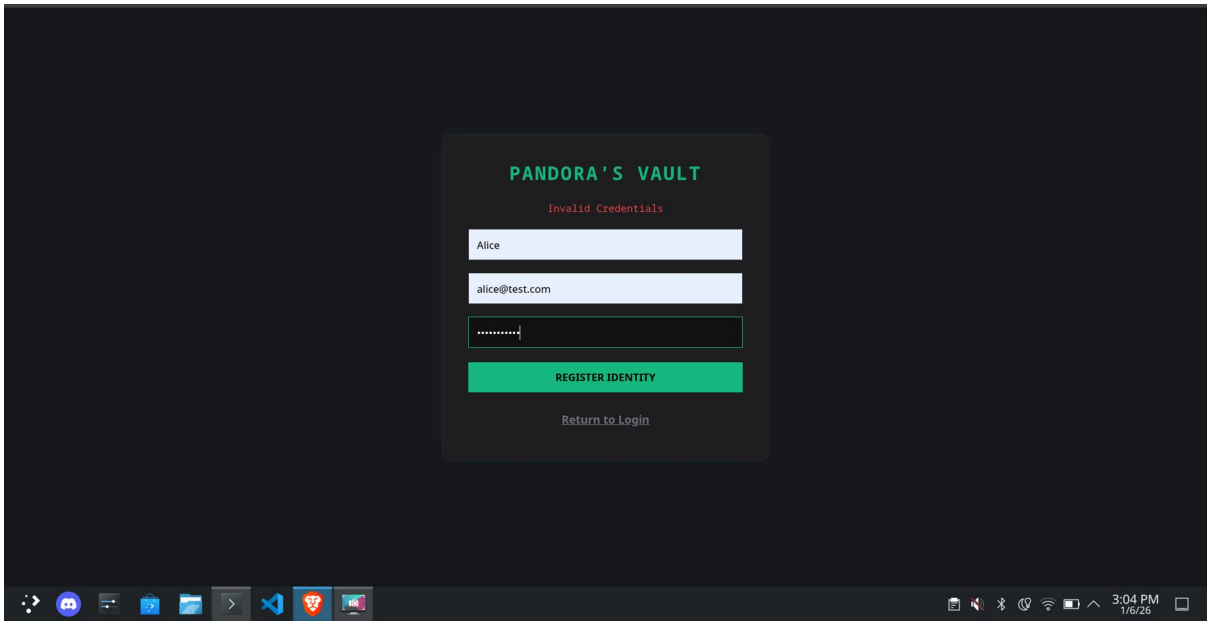
generate_totp: Generates the final OTP which the user gets. The intial loop converts the integer time to an 8-byte counter. This loop fills the array from back to front (big endian). Then a 20-byte hmac_result stores the current time signature and a key via a hash. Then it is truncated and masked to a 32-bit integer which is finally modulo by a million to get the 6-digit OTP users get.

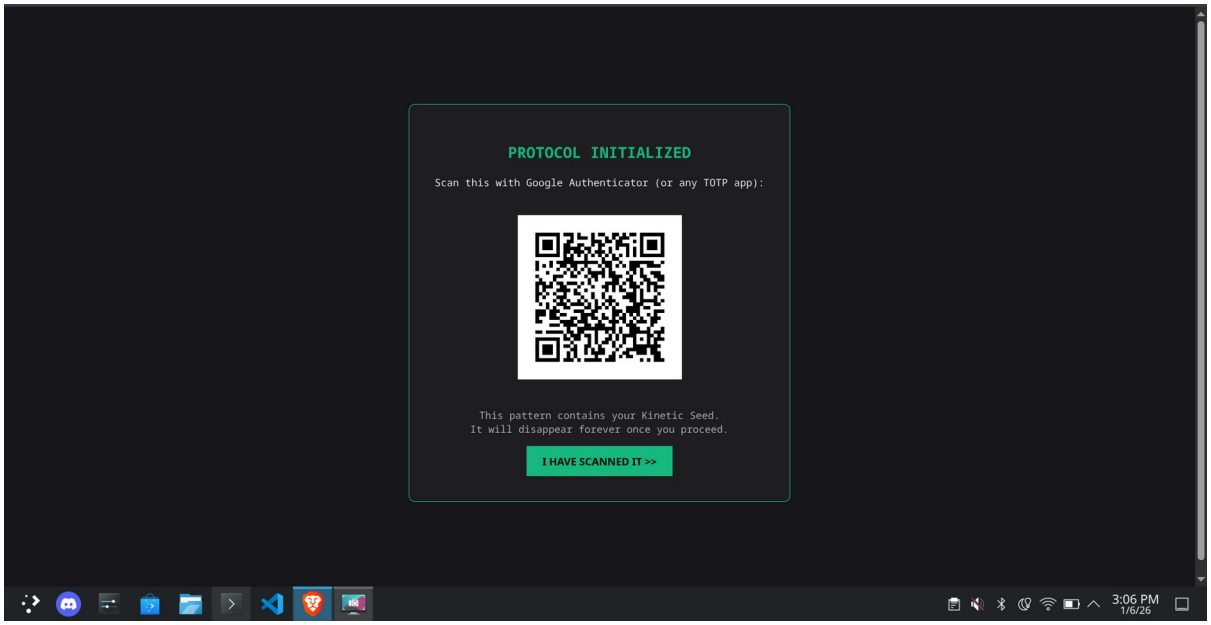
Demonstration x-D

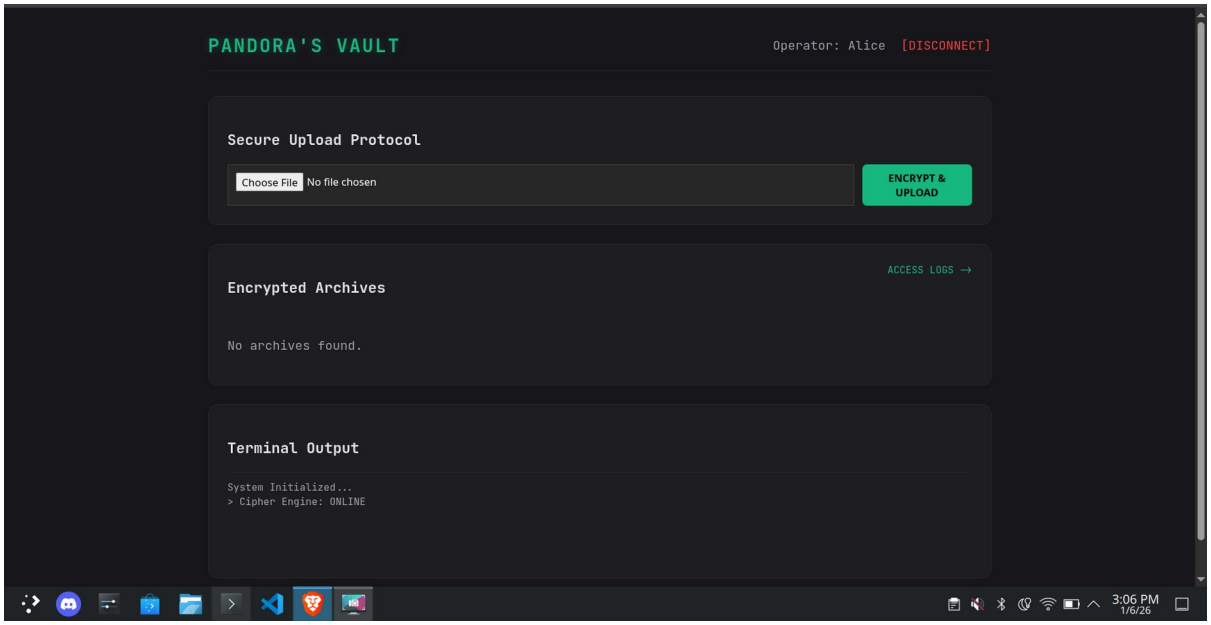
The next few slides will show a quick demonstration of how everything works out in the end.

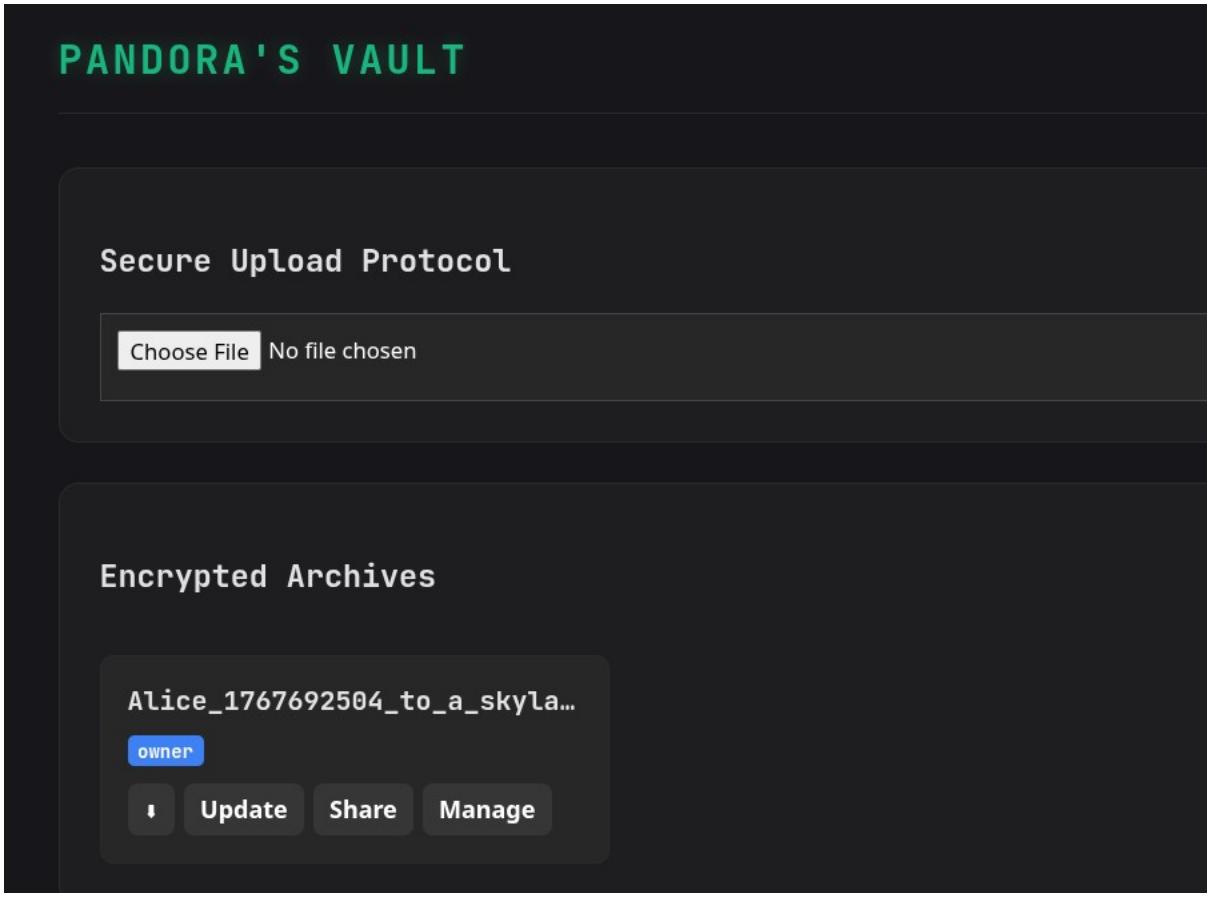
Signup and Login:

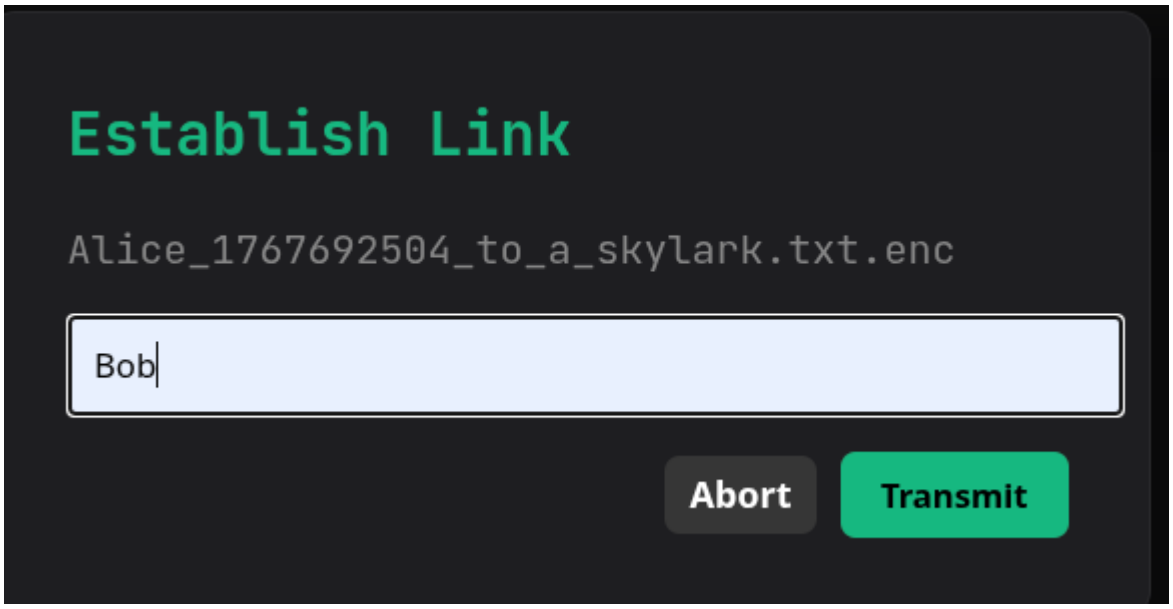





PROTOCOL INITIALIZED
Scan this with Google Authenticator (or any TOTP app):
This pattern contains your Kinetic Seed.
It will disappear forever once you proceed.
I HAVE SCANNED IT >>
3:06 PM
1/6/26


PANDORA'S VAULT
Operator: Alice [DISCONNECT]
Secure Upload Protocol
Choose File No file chosen
ENCRYPT & UPLOAD
ACCESS LOGS →
Encrypted Archives
No archives found.
Terminal Output
System Initialized...
> Cipher Engine: ONLINE
3:06 PM
1/6/26


PANDORA'S VAULT
Secure Upload Protocol
Choose File
No file chosen
Encrypted Archives
Alice_1767692504_to_a_skyla…
owner
Update
Share
Manage


Establish Link
Alice_1767692504_to_a_skylark.txt.enc
Bob
Abort
Transmit

Acknowledgments:

A few thanks are in order. Mostly, the information security division, the workshops and the help and resources they provided were amazing, especially Rachit sir, Sanjib Sir and Kartik Sir. Ayan sir is a great source of inspiration too. What he said during one of the workshops still stuck with me. Let's just say, the COTDs, unixit and the project (although tedious) were amazing.

Thanks and Regards

Ill-mannered-ambassador