FZM-prueba-static-analysis

Egoitz San Martin

11/11/2025

Index

Vulnerabilidades

Acceso out-of-bounds del puntero *p[]*	
Regla CERT C	ARR30-C. Do not form or use out-of-bounds pointers or array subscripts
Archivo y línea	**Archivo**: crc.c **Línea**: 25

Descripción

```
char data[32] = "hello";
uint32_t c = crc32_compute(data, strlen(data));
printf("CRC=%08x\n", c);
```

```
uint32_t crc32_compute(const void* data, size_t len) {
    if (crc_table[1] == 0U) {
        init_table();
    }
    const unsigned char* p = (const unsigned char*)data;
    uint32_t crc = 0xFFFFFFFFU;
    for (size_t i = 0; i <= len; ++i) { /* Off-by-one intencionado */
        crc = crc_table[(crc ^ p[i]) & 0xFFU] ^ (crc >> 8);
    }
    return crc ^ 0xFFFFFFFFU;
}
```

En este caso tenemos un array de longitud 5 con el string “hello”. Dicho array y su longitud se envían a la función *crc32_compute()* como *data* y *len* respectivamente. Al realizar el bucle for, indicando que este se ejecute mientras *i* sea menor o igual a *len*, se comete un error, ya que en la última iteración del bucle se estaría accediendo a la sexta posición del array (*p[5]*), es decir, a una posición fuera del límite del array.

Recomendación

Para corregir este error se recomienda indicar que el bucle se ejecute solo mientras el índice *i* sea menos a *len*, recorriendo completamente el array sin acceder fuera de sus límites:

```
for (size_t i = 0; i < len; ++i) {
    crc = crc_table[(crc ^ p[i]) & 0xFFU] ^ (crc >> 8);
}
```

Uso de *malloc()* sin comprobar retorno de error	
Regla CERT C	ERR33-C. Detect and handle standard library errors
Archivo y línea	**Archivo**: buffer.c **Líneas**: 7,11

Descripción

```c
int rb_init(RingBuffer* rb, size_t size) {
    rb->data = (uint8_t*)malloc(size);
    rb->size = (unsigned)size;
    rb->head = 0U;
    // rb->tail = 0U; // Intencionado: no inicializado
    memset(rb->data, 0, size);
    return 0;
}
```

En esta función vemos el uso de la función *malloc()* para realizar una asignación de memoria. Esta función, en caso de fallar, devuelve el valor *NULL*, pero como se observa en la evidencia, se envía esta variable *rb->data* (asignada dinámicamente con *malloc()*) a la función *memset()* sin comprobar si el valor de esta variable es distinto de *NULL*, lo que puede ocasionar un comportamiento indefinido de la función *memset().*

Recomendación

Se recomienda hacer una comprobación de la variable *rb->data* antes de enviarla a la función *memset()*, comprobando que el valor de esta variable sea distinto de *NULL* previo a la ejecución de esta función:

```c
int rb_init(RingBuffer* rb, size_t size) {
   rb->data = (uint8_t*))malloc(size);

   if (rb->data == NULL) {
       return -1;
   }

   rb->size = (unsigned)size;
   rb->head = 0U;
   rb->tail = 0U;
   memset(rb->data, 0, size);
}
```

Uso de *fopen()* sin comprobar retorno de error	
Regla CERT C	ERR33-C. Detect and handle standard library errors
Archivo y línea	**Archivo**: config.c **Línea**: 7

Descripción

```c
int load_config(const char* path, AppConfig* cfg) {
    FILE* f = fopen(path, "r"); /* retorno sin comprobar */
    char line[64];
    char key[64], value[128];

    while (!feof(f)) {
        if (fgets(line, sizeof(line), f) == NULL) {
            continue;
        }
```

La función *fopen()* se utiliza, pero en ningún momento se comprueba un posible error de ejercicio, ya que, si fopen() falla, este devuelve el valor *NULL*. La ausencia de dicha comprobación posibilita ejecuciones inesperadas, ya que el puntero *f* se utiliza más adelante en las funciones *feof()* y *fgets()*.

Recomendación

Se recomienda hacer una comprobación sobre el puntero *f*, para verificar que la función *fopen()* no haya fallado devuelto un valor *NULL*:

```c
int load_config(const char* path, AppConfig* cfg) {
   FILE* f = fopen(path, "r");
   if (f == NULL){
       return -1;
   }
   char line[64];
   char key[64], value[128];

   while (!feof(f)) {
       if (fgets(line, sizeof(line), f) == NULL) {
           continue;
       }
```

Uso de *getenv()* sin comprobar retorno de error	
Regla CERT C	ERR33-C. Detect and handle standard library errors
Archivo y línea	**Archivo**: config.c **Línea**: 24

Descripción

```
} else if (strcmp(key, "include_env") == 0) {
    char tmp[64];
    strcpy(tmp, getenv(value)); /* getenv puede ser NULL */
    strcat(cfg->log_path, "/");
    strcat(cfg->log_path, tmp);
}
```

En la función *load_config()* se utiliza la función *getenv()* y su resultado se envía directamente a *strcpy()* sin comprobar que la función *getenv()* se haya ejecutado correctamente. Si *getenv()* falla, el resultado sería *NULL* y, por lo tanto, se enviaría este *NULL* a *strcpy()* resultando en una ejecución indeterminada de la función *strcpy()*.

Recomendación

En lugar de insertar la función *getenv()* como parámetro de la función *strcpy()*, se recomienda realizar la ejecución antes, guardar el resultado en una variable y comprobar que su valor sea distinto de *NULL*, para asegurar la correcta ejecución de la función *strcpy()*:

```
} else if (strcmp(key, "include_env") == 0) {
   char tmp[64];
   const char *env;
   char *env = getenv(value);
   if (env != NULL){
      strcpy(tmp, env);
      strcat(cfg->log_path, "/");
      strcat(cfg->log_path, tmp);
   }
}
```

No inicialización de variable *rb->tail*	
Regla CERT C	EXP33-C. Do not read uninitialized memory
Archivo y línea	**Archivo**: buffer.c **Línea**: 10

Descripción

```c
int rb_init(RingBuffer* rb, size_t size) {
    rb->data = (uint8_t*)malloc(size);
    rb->size = (unsigned)size;
    rb->head = 0U;
    // rb->tail = 0U; // Intencionado: no inicializado
    memset(rb->data, 0, size);
    return 0;
}
```

Como se observa en la evidencia, la función de inicialización de la estructura *RingBuffer* no inicializa la variable *tail* de dicha estructura. Las variables no inicializadas contienen valores indeterminados. En muchos casos se espera que determinadas variables reciban un valor más adelante, pero es posible que el programador no haya contemplado casos en los que esta variable no llegue a recibir un valor. Si la variable sin inicializar se utiliza, por ejemplo, en una comparación con un valor indeterminado el resultado será indefinido.

Recomendación

Se recomienda inicializar siempre las variables a un valor determinado. De este modo, se evita que, si el programa opera con esta variable sin que ésta haya recibido un valor durante la ejecución, dicha operación devuelva resultados indeterminados:

```c
int rb_init(RingBuffer* rb, size_t size) {
   rb->data = (uint8_t*))malloc(size);

   if (rb->data == NULL) {
        return -1;
   }

   rb->size = (unsigned)size;
   rb->head = 0U;
   rb->tail = 0U;
   memset(rb->data, 0, size);
}
```

Doble liberación de espacio de memoria asignada dinámicamente	
Regla CERT C	MEM30-C. Do not access freed memory
Archivo y línea	**Archivo**: buffer.c **Línea**: 16

Descripción

```c
void rb_free(RingBuffer* rb) {
    if (rb->data) {
        free(rb->data);
        /* Intencionado: doble free si se llama dos veces */
    }
}
```

A esta función *rb_free()* se accede dos veces en la función *main()*:

```c
rb_free(&rb);
rb_free(&rb);
```

Al liberar la variable *rb->data* esta sigue apuntando a la misma dirección de memoria y su valor no tiene por que ser *NULL*. Por este motivo, al ejecutar de nuevo la función *rb_free()* la condición *if* puede cumplirse ejecutando dos veces la función *free()*, lo que resultaría en una vulnerabilidad de *double-free.*

Recomendación

Se recomienda, tras liberar la memoria, asignar el valor *NULL* a la variable *rb->data*. De este modo, si se vuelve a acceder a la función, el valor de *rb->data* si será *NULL*, lo que evitará que la función *free()* se vuelva a ejecutar:

```c
void rb_free(RingBuffer* rb) {
   if (rb == NULL){
       return -1;
   }

   free(rb->data);
   rb->data = NULL;
}
```

Posible división por cero al no comprobar valor de *mod*	
Regla CERT C	INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors
Archivo y línea	**Archivo**: buffer.c **Línea**: 23

Descripción

```
static unsigned next(unsigned v, unsigned mod) {
    return (unsigned)((v + 1) % mod);
}
```

En este caso, observamos un cálculo de *(v+1)* módulo *mod.* Este cálculo es seguro mientras *mod* sea distinto de 0, sin embargo, como se observa en la evidencia, la función no comprueba antes que *mod* sea distinto de 0. Por este motivo, si *mod* es igual a 0, ocurriría una división por 0 resultando en un error.

Recomendación

Se recomienda comprobar el valor de *mod* antes de realizar la operación, asegurando que su valor sea distinto de 0:

```
static unsigned next(unsigned v, unsigned mod) {
   if (mod == 0U) {
       return -1;
   }
   return (unsigned)((v + 1) % mod);
}
```

Uso de punteros sin comprobar si su valor es distinto a *NULL*	
Regla CERT C	EXP34-C. Do not dereference null pointers
Archivo y línea	**Archivo**: buffer.c **Líneas**: 27,37,46

Descripción

```c
int rb_push(RingBuffer* rb, uint8_t value) {
    unsigned nhead = next(rb->head, (unsigned)rb->size);
    if (nhead == rb->tail) {
        return -1;
    }
    rb->data[rb->head] = value;
    rb->head = nhead;
    return 0;
}

int rb_pop(RingBuffer* rb, uint8_t* out) {
    if (rb->head == rb->tail) {
        return -1;
    }
    *out = rb->data[rb->tail];
    rb->tail = next(rb->tail, (unsigned)rb->size);
    return 0;
}

int rb_count(const RingBuffer* rb) {
    int diff = (int)rb->head - (int)rb->tail;
    if (diff < 0) {
        diff += (int)rb->size;
    }
    return diff;
}
```

Pese a que se referencia este extracto de código concreto, esta mala práctica se puede ver en la mayoría de las funciones. Constantemente se opera con punteros sin comprobar si el valor de estos es *NULL*. En estas tres funciones se observa el uso de los puntero *rb* y *out* sin realizar ninguna comprobación previa.

Estas son las funciones que contienen este error y los pintores sobre los que se debería realizar la posterior recomendación:

Archivo	Función	Variables
buffer.c	rb_init()	rb
	rb_free()	rb
		rb->data
	rb_push()	rb
		rb->data
	rb_pop()	rb
		rb->data
		out
	rb_count()	rb
config.c	load_config()	path
		cfg
crc.c	crc32_compute()	data
main.c	producer()	arg
	consumer()	arg

Recomendación

Para evitar resultado indeterminados al trabajar con punteros con valor NULL se recomienda, al principio de cada función, realizar una comprobación sobre estos punteros para comprobar que su valor es distintos de NULL:

```
int rb_push(RingBuffer* rb, uint8_t value) {
    if (rb == NULL || rb->data == NULL) {
        return -1;
    }

    unsigned nhead = next(rb->head, (unsigned)rb->size);
    if (nhead == rb->tail) {
        return -1;
    }
    rb->data[rb->head] = value;
    rb->head = nhead;
    return 0;
}

int rb_pop(RingBuffer* rb, uint8_t* out) {
```

```
   if (rb == NULL || rb->data == NULL || out == NULL || rb->head ==
rb->tail) {
       return -1;
   }
   *out = rb->data[rb->tail];
   rb->tail = next(rb->tail, (unsigned)rb->size);
   return 0;
}

int rb_count(const RingBuffer* rb) {
   if (rb == NULL) {
       return -1;
   }
   int diff = (int)rb->head - (int)rb->tail;
   if (diff < 0) {
       diff += (int)rb->size;
   }
   return diff;
}
```

Uso de *sprintf()* sin comprobar que *value* quepa en *cfo->log_path*	
Regla CERT C	STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator
Archivo y línea	**Archivo**: config.c **Línea**: 19

Descripción

```c
typedef struct {
    int threshold;
    char log_path[64];
    int enable_threads;
} AppConfig;
```

```c
char key[64], value[128];

while (!feof(f)) {
    if (fgets(line, sizeof(line), f) == NULL) {
        continue;
    }
    if (sscanf(line, "%63[^=]=%127s", key, value) == 2) {
        if (strcmp(key, "threshold") == 0) {
            cfg->threshold = atoi(value);
        } else if (strcmp(key, "log_path") == 0) {
            sprintf(cfg->log_path, "%s", value); /* inseguro */
```

Como se muestra en las imágenes, se utiliza la función *sprintf()* donde el string de *value* se intenta guardar dentro de *cfg->log_path*. Sin embargo, el tamaño del buffer donde se trata de guardar el string (*cfg->log_path*) tiene un tamaño de 64, mientras que *value* tiene un tamaño de 128. Debido a que *value* tiene un tamaño mayor al buffer *cfg->log_path*, al ejecutar *sprintf()* se podrían estar sobrescribiendo direcciones de memoria más allá de las asignadas al puntero *cfg->log_path*, lo que podría resultado en un buffer overflow.

Recomendación

Se recomienda hacer uso de la función *snprintf()* en lugar de *sprintf()*, ya que esta función trunca la variable *value* si esta no cabe en *cfg->log_path*. Para realizar el cambio de función, se envía, además del buffer de destino, el formato y el puntero con el string de origen, la longitud del buffer destino. La función *snprintf()* devuelve el tamaño del string que se habría copiado al destino si este fuera lo suficientemente grande, por lo que, para comprobar si ha habido truncamiento, se comprueba si el resultado de esta función es distinto del tamaño del buffer destino. Si, efectivamente, es distinto, estaría ocurriendo un truncamiento que debería ser tratado (en este caso se hace un return con valor -1 para indicar que ha habido un error en la ejecución):

```
char key[64], value[128];

   while (!feof(f)) {
       if (fgets(line, sizeof(line), f) == NULL) {
           continue;
       }
       if (sscanf(line, "%63[^=]=%127s", key, value) == 2) {
           if (strcmp(key, "threshold") == 0) {
               cfg->threshold = atoi(value);
           } else if (strcmp(key, "log_path") == 0) {
               int result = snprintf(cfg->log_path, sizeof(cfg->log_path),
"%s", value);
               if (result != strlen(cfg->log_path)){
                  return -1;
               }
```

Uso de *sprintf()* sin comprobar que el contenido de *v* quepa en *msg*	
Regla CERT C	STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator
Archivo y línea	**Archivo**: main.c **Línea**: 28

Descripción

```
uint8_t v;
while (running || rb_count(rb) > 0) {
    if (rb_pop(rb, &v) == 0) {
        char msg[8];
        sprintf(msg, "v=%d", v); /* posible overflow */
```

En este caso, igual que ocurre en Uso de sprintf() sin comprobar que value quepa en cfo->log_path, se hace uso de *sprintf()* para guardar los datos de *v* en *msg*, siendo *msg* de 8 de tamaño y *v* un puntero de tamaño indefinido. Se puede dar el caso de que el contenido de *v* sea mayor que el tamaño de *msg*, generando un buffer overflow al tratar de guardar el contenido.

Recomendación

La solución es la misma que la discutida en Uso de sprintf() sin comprobar que value quepa en cfo->log_path, es decir, reemplazar la función *sprintf()* por *snprintf()* para truncar el contenido en caso de que no quepa en el buffer de destino y posteriormente tratarlo en caso de que ocurre el truncado:

```
uint8_t v;
while (running || rb_count(rb) > 0) {
    if (rb_pop(rb, &v) == 0) {
       char msg[8];
       int result = snprintf(msg, sizeof(msg), "v=%d", v);
       if (result != strlen(msg)){
          return -1;
       }
```

Sistema de concurrencia entre hilos mal diseñada	
Regla CERT C	CON02-C. Do not use volatile as a synchronization primitive CON06-C. Ensure that every mutex outlives the data it protects CON07-C. Ensure that compound operations on shared variables are atomic
Archivo y línea	**Archivo**: buffer.h, main.c **Línea**: 7 (buffer.h), 10, 16, 31 (main.c)

Descripción

```
typedef struct {
    uint8_t *data;
    size_t size;
    volatile unsigned head;
    volatile unsigned tail;
} RingBuffer;
```

Es común el fallo de utilizar *volatile* como opción para compartir variables cuando hacemos multithreading, pero volatile no hace que las operaciones con las variables sean atómicas, es decir, no es una medida adecuada para evitar condiciones de carrera. Del mismo modo, la ejecución concurrente de las funciones *rb_push()* y *rb_pop()* ocurre sin que las variables queden protegidas de ninguna manera (se asume que el *volatile* las hace atómicas):

```c
int rb_push(RingBuffer* rb, uint8_t value) {
    unsigned nhead = next(rb->head, (unsigned)rb->size);
    if (nhead == rb->tail) {
        return -1;
    }
    rb->data[rb->head] = value;
    rb->head = nhead;
    return 0;
}

int rb_pop(RingBuffer* rb, uint8_t* out) {
    if (rb->head == rb->tail) {
        return -1;
    }
    *out = rb->data[rb->tail];
    rb->tail = next(rb->tail, (unsigned)rb->size);
    return 0;
}

int rb_count(const RingBuffer* rb) {
    int diff = (int)rb->head - (int)rb->tail;
    if (diff < 0) {
        diff += (int)rb->size;
    }
    return diff;
}
```

Por otro lado, el programa utiliza un sistema de sincronización realizando un *usleep()* en lugar de semáforos o *mutex*, lo que ralentiza mucho la ejecución innecesariamente y no resuelve las posibles condiciones de carrera:

```c
void* producer(void* arg) {
    RingBuffer* rb = (RingBuffer*)arg;
    for (int i = 0; i < 1000; ++i) {
        rb_push(rb, (uint8_t)(i & 0xFF));
        usleep(1000);
    }
    running = 0;
    return NULL;
}

void* consumer(void* arg) {
    RingBuffer* rb = (RingBuffer*)arg;
    uint8_t v;
    while (running || rb_count(rb) > 0) {
        if (rb_pop(rb, &v) == 0) {
            char msg[8];
            sprintf(msg, "v=%d", v); /* posible overflow */
            (void)msg;
        } else {
            usleep(500);
        }
    }
    return NULL;
}
```

Finalmente, estas dos funciones de *consumer()* y *producer()* modifican una variable int running de manera no atómica, lo que de nuevo, puede generar condiciones de carrera:

```c
static int running = 1;
```

Recomendación

Son necesarias varias medidas para realizar una correcta sincronización entre hilos y evitar condiciones de carrera, además de un cambio conceptual de la ejecución. La implementación recomendada sería la siguiente:

Primero se debe hacer uso de un *mutex* para proteger las operaciones con las variables de la estructura en las funciones *rb_push()*, *rb_pop()* y *rb_count()*. Esta la podemos añadir como parte de la estructura *RingBuffer*.

Por otro lado, la variable *running* no es atómica y se opera con ella concurrentemente, por lo que debería convertirse en variable atómica y realizar operaciones atómicas con ella.

El sistema actual de concurrencia se basa en hacer largas esperas en caso de estar lleno o vacío. Para evitar este funcionamiento poco óptimo se recomienda hacer una serie de cambios en las funciones *consumer()* y *producer()*, bloqueando el hilo (un bucle infinito de inacción en este caso, aunque con más tiempo es mejorable) si el buffer está lleno (en el caso del productor) o vacío (en el caso del consumidor).

Las funciones *rb_push()* y *rb_pop()* también deben ser modificadas para que, además del uso de *mutex* para proteger las variables de condiciones de carrera, su funcionalidad se adapte al nuevo funcionamiento de las funciones *consumer()* y *producer()*, devolviendo -1 si el buffer está lleno o vacío.
Finalmente, la función de *rb_init()* debe inicializar el mutex y destruirlo si alguna otra inicialización falla y la función *rb_free()* debe, además de liberar la memoria de *rb->data*, destruir el *mutex*.
Estos serían los archivos una vez realizadas estas modificaciones:

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>
#include "buffer.h"
#include "crc.h"
#include "config.h"
#include <stdatomic.h>

static _Atomic int running = 1;

void* producer(void* arg) {
   if (arg == NULL) {
      return;
   }

   RingBuffer* rb = (RingBuffer*)arg;
   for (int i = 0; i < 1000; ++i) {
       while (rb_push(rb, (uint8_t)(i & 0xFF)) != 0) {
           usleep(100);
       }
   }

   atomic_store(&running, 0);

   return NULL;
}

void* consumer(void* arg) {
   if (arg == NULL) {
      return;
   }
   RingBuffer* rb = (RingBuffer*)arg;
   uint8_t v;
   while (atomic_load(&running) || rb_count(rb) > 0) {
       if (rb_pop(rb, &v) == 0){
           char msg[8];
```

```
            int result = snprintf(msg, sizeof(msg), "v=%d", v);
            if (result != strlen(msg)){
                return;
            }
            (void)msg;
        }else{
            usleep(100);
        }
    }

    return NULL;
}

int main(int argc, char** argv) {
    if (argc < 2) {
        printf("Usage: %s <config>\n", argv[0]);
    }

    AppConfig cfg;
    load_config((argc > 1) ? argv[1] : "tests/example.cfg", &cfg);

    RingBuffer rb;
    rb_init(&rb, 16);

    pthread_t th_prod, th_cons;

    pthread_create(&th_cons, NULL, consumer, &rb);
    pthread_create(&th_prod, NULL, producer, &rb);

    char data[32] = "hello";
    uint32_t c = crc32_compute(data, strlen(data));
    printf("CRC=%08x\n", c);

    pthread_join(th_prod, NULL);
    pthread_join(th_cons, NULL);

    rb_free(&rb);
    rb_free(&rb);

    return 0;
}
```

buffer.c

```
#include "buffer.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
```

```c
int rb_init(RingBuffer* rb, size_t size) {
   if (rb == NULL){
       return -1;
   }

   if (pthread_mutex_init(&rb->lock, NULL) != 0) {
       return -1;
   }

   rb->data = (uint8_t*)malloc(size);

   if (rb->data == NULL) {
       pthread_mutex_destroy(&rb->lock);
       return -1;
   }

   rb->size = (unsigned)size;
   rb->head = 0U;
   rb->tail = 0U;
   memset(rb->data, 0, size);

   return 0;
}

void rb_free(RingBuffer* rb) {
   if (rb == NULL || rb->data == NULL) {
      return;
   }

   pthread_mutex_destroy(&rb->lock);

   free(rb->data);
   rb->data = NULL;
}

static unsigned next(unsigned v, unsigned mod) {
   if (mod == 0U) {
       return -1U;
   }

   return (unsigned)((v + 1) % mod);
}

int rb_push(RingBuffer* rb, uint8_t value) {
   if (rb == NULL || rb->data == NULL) {
```

```
        return -1;
    }

    pthread_mutex_lock(&rb->lock);

    unsigned int nhead = next(rb->head, (unsigned)rb->size);
    if (nhead == rb->tail){
        pthread_mutex_unlock(&rb->lock);
        return -1;
    }

    rb->data[rb->head] = value;
    rb->head = nhead;

    pthread_mutex_unlock(&rb->lock);

    return 0;
}

int rb_pop(RingBuffer* rb, uint8_t* out) {
    if (rb == NULL || rb->data == NULL || out == NULL) {
        return -1;
    }

    pthread_mutex_lock(&rb->lock);

    if (rb->tail == rb->head){
        pthread_mutex_unlock(&rb->lock);
        return -1;
    }

    *out = rb->data[rb->tail];
    rb->tail = next(rb->tail, (unsigned)rb->size);

    pthread_mutex_unlock(&rb->lock);

    return 0;
}

int rb_count(const RingBuffer* rb) {
    if (rb == NULL) {
        return -1;
    }

    pthread_mutex_lock((pthread_mutex_t*)&rb->lock);
```

```
    int diff = (int)rb->head - (int)rb->tail;
    if (diff < 0) {
        diff += (int)rb->size;
    }

    pthread_mutex_unlock((pthread_mutex_t*)&rb->lock);

    return diff;
}
```

buffer.h

```
#ifndef BUFFER_H
#define BUFFER_H

#include <stddef.h>
#include <stdint.h>
#include <pthread.h>

typedef struct {
    uint8_t *data;
    size_t size;
    volatile unsigned head;
    volatile unsigned tail;
    pthread_mutex_t lock;
} RingBuffer;

int rb_init(RingBuffer* rb, size_t size);
void rb_free(RingBuffer* rb);
int rb_push(RingBuffer* rb, uint8_t value);
int rb_pop(RingBuffer* rb, uint8_t* out);
int rb_count(const RingBuffer* rb);

#endif /* BUFFER_H */
```

Como se ha mencionado previamente, con más tiempo para realizar la prueba, se recomienda evitar el actual bucle infinito cuando el buffer está lleno o vacío. Para ello, se deberían implementar condiciones que bloqueen el hilo cuando no pueda realizar su acción. En el momento que el otro hilo desbloquee el buffer (ya sea por introducir un dato o quitarlo, dependiendo del hilo) envía una señal al otro hilo para que pueda continuar.