Created By: **Eduardo Argueta Cantizzano**
Vulnerability: Pthreads Mutex Synchronization attack

Pthreads Mutex Synchronization.

A Pthreads Mutex Synchronization attack refers to a security vulnerability that exploits weaknesses in the usage of mutexes (mutual exclusion locks) within programs that utilize the POSIX Threads (Pthreads) library for multithreading synchronization.

Here’s how such attacks can occur and their implications:

1. **Incorrect Mutex Usage**: If mutexes are not properly initialized, locked, unlocked, or destroyed, it can lead to race conditions or deadlocks. These conditions can potentially be exploited by an attacker to gain unauthorized access to shared resources or to disrupt the program's intended behavior.
2. **Race Conditions**: Improper synchronization using mutexes can create race conditions, where the outcome of operations depends on the sequence or timing of thread execution. Attackers may attempt to manipulate this timing to achieve unintended outcomes, such as accessing resources concurrently when they should not be accessible.
3. **Deadlocks**: Incorrect use of mutexes can also lead to deadlocks, where threads become stuck waiting indefinitely for resources that are held by other threads.

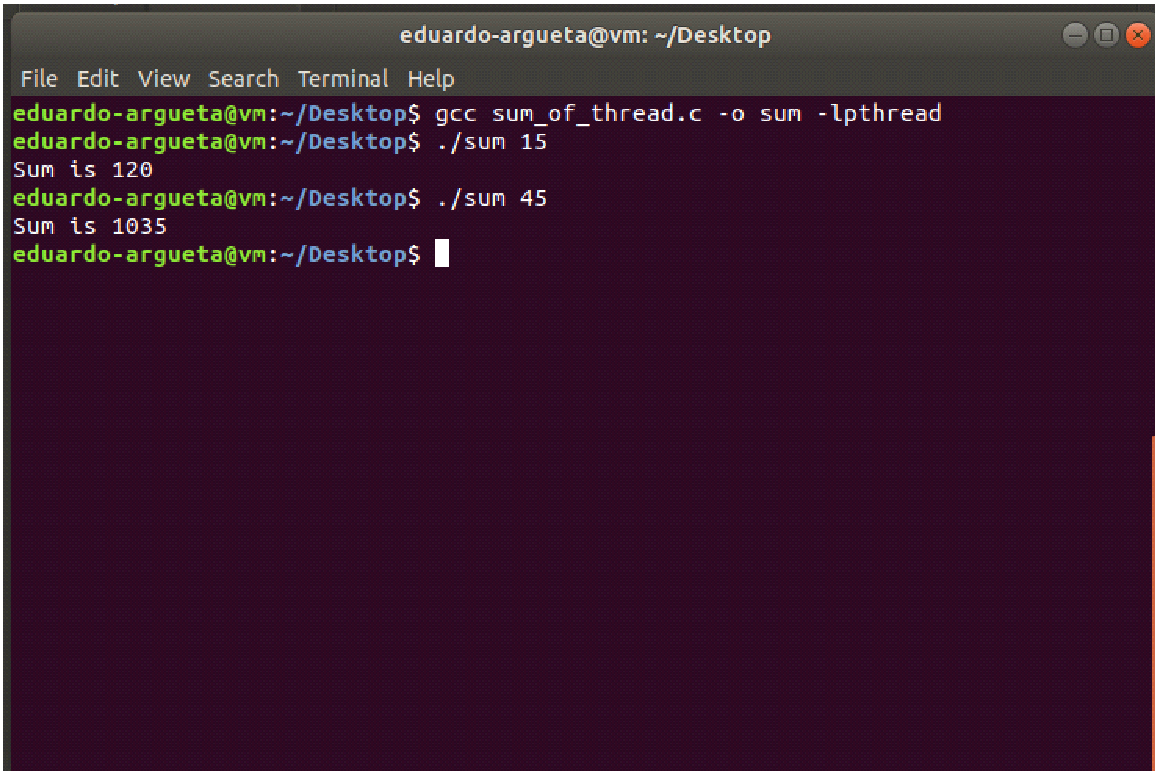
This can be exploited to cause denial of service (DoS) attacks or to disrupt the normal functioning of the application.

4. **Exploitation**: In a Pthreads Mutex Synchronization attack, an attacker might attempt to force a race condition or deadlock scenario by manipulating the timing or sequence of thread operations. This could potentially lead to data corruption, unauthorized data access, or other security vulnerabilities depending on the context of the application.

Creating a thread.

I am creating a thread. We have passed a thread function called *sum_runner* which calculates the sum of number till specified number. It calculates the sum and stores it in the global variable *sum*.

```
Open ▾   Save
        for(long long i = 0; i <= limit; i++) {
        sum += i;
        }

        //sum is a global variable, so other threads can access

        pthread_exit(0);
}

int main(int argc, char * argv[])
{
        if(argc < 2) {
                printf("Usage: %s <num>\n", argv[0]);
                exit(-1);
        }

        long long limit = atoll(argv[1]);

        //Thread ID
        pthread_t tid;

        //Create attributes
        pthread_attr_t attr;
        pthread_attr_init(&attr);

        pthread_create(&tid, &attr, sum_runner, &limit);

        pthread_join(tid, NULL);
        printf("Sum is %lld\n", sum);
}
```



We see how to use mutex locks in threds to synchronize them. will see in the following types of modifications codes.

Modification # 1

In this modification, we have created a fucntion *counting_function()* which keeps on adding the number *offset* *NUM_LOOP* times. We have set *NUM_LOOP* a very large number i.e. 5000000000. Now we will see how much time it takes to iterate this loop which no thread is used.

```
Open ▾                                                   Save
#include <stdio.h>

#define NUM_LOOPS 500000000
long long sum = 0;

void counting_function(int offset)
{
        for(int i=0; i<NUM_LOOPS; i++) {
                sum += offset;
        }
}

int main(void)
{
        counting_function(1);
        printf("Sum = %lld\n", sum);
        return 0;
}
```

```
eduardo-argueta@vm: ~/Desktop
File  Edit  View  Search  Terminal  Help
eduardo-argueta@vm:~/Desktop$ gcc mod1.c -o mod1
eduardo-argueta@vm:~/Desktop$ ./mod1
Sum = 500000000
eduardo-argueta@vm:~/Desktop$ time ./mod1
Sum = 500000000

real    0m1.243s
user    0m1.238s
sys     0m0.004s
eduardo-argueta@vm:~/Desktop$
```

As we can see, it took 1243 milliseconds to iterate this without threads.

Modification # 2

In this modification, we have just call that *counting_function()* two times to observe the time it takes.

```c
#include <stdio.h>

#define NUM_LOOPS 500000000
long long sum = 0;

void counting_function(int offset)
{
        for(int i=0; i<NUM_LOOPS; i++) {
                sum += offset;
        }
}

int main(void)
{
        counting_function(1);
        counting_function(-1);
        printf("Sum = %lld\n", sum);
        return 0;
}
```

```
eduardo-argueta@vm:~/Desktop$ gcc mod2.c -o mod2
eduardo-argueta@vm:~/Desktop$ ./mod2
Sum = 0
eduardo-argueta@vm:~/Desktop$ time ./mod2
Sum = 0

real    0m2.278s
user    0m2.271s
sys     0m0.004s
eduardo-argueta@vm:~/Desktop$ time ./mod2
Sum = 0

real    0m3.376s
user    0m3.361s
sys     0m0.005s
eduardo-argueta@vm:~/Desktop$ time ./mod2
Sum = 0

real    0m3.433s
user    0m3.423s
sys     0m0.009s
eduardo-argueta@vm:~/Desktop$
```

The time got almost double and we haven't used threads yet.

Modification # 3

Now in this modification, we have created a thread for that function or loop to iterate *NUM_LOOP* times. We have passed the *offset* as a thread argument.

```
Open ▾                                                          Save
#include <stdio.h>
#include <pthread.h>

#define NUM_LOOPS 500000000
long long sum = 0;

void * counting_function(void * arg)
{
        int offset = *(int *) arg;

        for(int i=0; i<NUM_LOOPS; i++) {
                sum += offset;
        }

        pthread_exit(NULL);
}

int main(void)
{
        //counting_function(1);

        pthread_t id1;
        int offset1 = 1;
        pthread_create(&id1, NULL, counting_function, &offset1);
        pthread_join(id1, NULL);

        printf("Sum = %lld\n", sum);
}
```

```
eduardo-argueta@vm: ~/Desktop
File Edit View Search Terminal Help
eduardo-argueta@vm:~/Desktop$ gcc mod3.c -o mod3 -lpthread
eduardo-argueta@vm:~/Desktop$ ./mod3
Sum = 500000000
eduardo-argueta@vm:~/Desktop$ time ./mod3
Sum = 500000000

real    0m1.578s
user    0m1.576s
sys     0m0.000s
eduardo-argueta@vm:~/Desktop$ time ./mod3
Sum = 500000000

real    0m1.848s
user    0m1.838s
sys     0m0.008s
eduardo-argueta@vm:~/Desktop$ time ./mod3
Sum = 500000000

real    0m1.793s
user    0m1.785s
sys     0m0.005s
eduardo-argueta@vm:~/Desktop$
```

We can observe that it didn't make a difference in time. It's of no use to use 1 thread to iterate that loop.

Modification # 4

Hence, in this modification, we have created 2 threads to observe the time. Each thread is calling the function - *counting_function()*.

```c
#define NUM_LOOPS 500000000
long long sum = 0;

void * counting_function(void * arg)
{
        int offset = *(int *) arg;

        for(int i=0; i<NUM_LOOPS; i++) {
                sum += offset;
        }

        pthread_exit(NULL);
}

int main(void)
{
        //counting_function(1);

        pthread_t id1;
        int offset1 = 1;
        pthread_create(&id1, NULL, counting_function, &offset1);
        pthread_join(id1, NULL);

        pthread_t id2;
        int offset2 = -1;
        pthread_create(&id2, NULL, counting_function, &offset2);
        pthread_join(id2, NULL);

        printf("Sum = %lld\n", sum);
}
```

```
eduardo-argueta@vm: ~/Desktop
File  Edit  View  Search  Terminal  Help
eduardo-argueta@vm:~/Desktop$ gcc mod4.c -o mod4 -lpthread
eduardo-argueta@vm:~/Desktop$ ./mod4
Sum = 0
eduardo-argueta@vm:~/Desktop$ time ./mod4
Sum = 0

real    0m2.931s
user    0m2.924s
sys     0m0.005s
eduardo-argueta@vm:~/Desktop$ time ./mod4
Sum = 0

real    0m2.890s
user    0m2.875s
sys     0m0.013s
eduardo-argueta@vm:~/Desktop$ time ./mod4
Sum = 0

real    0m2.920s
user    0m2.899s
sys     0m0.009s
eduardo-argueta@vm:~/Desktop$
```

As can be seen, it still didn't make a difference. It seems same as we have called the function 2 times like we did in modification#2. The reason is that, we, first of all, run thread1 and then waits until thread1 has completed its execution. As soon as thread1 completes its execution, we start running the thread2. That's why it didn't make any difference in time.

Modification # 5

Therefore, in this modification, we aren't waiting for thread1 to complete its execution, instead as soon as thread1 starts running, we start running the thread2. So, these 2 threads, thread1 and thread2, are running parallel to minimize the time.

```c
void * counting_function(void * arg)
{
        int offset = *(int *) arg;

        for(int i=0; i<NUM_LOOPS; i++) {
                sum += offset;
        }

        pthread_exit(NULL);
}

int main(void)
{
        //counting_function(1);

        //Spawn threads
        pthread_t id1;
        int offset1 = 1;
        pthread_create(&id1, NULL, counting_function, &offset1);

        pthread_t id2;
        int offset2 = -1;
        pthread_create(&id2, NULL, counting_function, &offset2);

        //Wait for threads
        pthread_join(id1, NULL);
        pthread_join(id2, NULL);

        printf("Sum = %lld\n", sum);
}
```

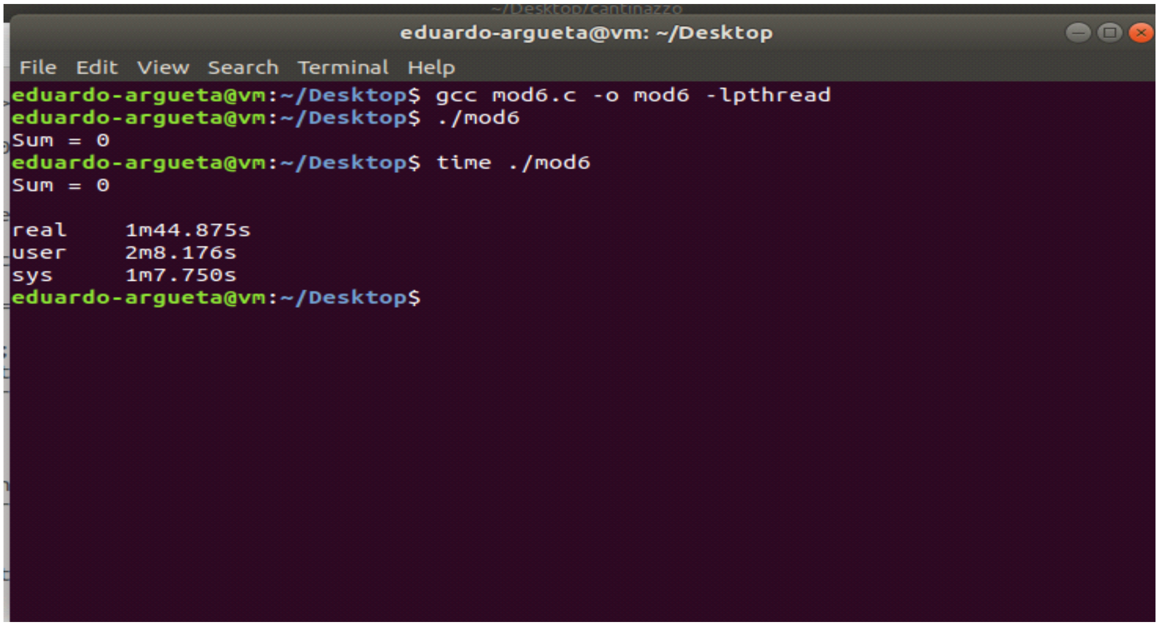
```
eduardo-argueta@vm: ~/Desktop
File  Edit  View  Search  Terminal  Help
eduardo-argueta@vm:~/Desktop$ gcc mod5.c -o mod5 -lpthread
eduardo-argueta@vm:~/Desktop$ ./mod5
Sum = -5648630
eduardo-argueta@vm:~/Desktop$ time ./mod5
Sum = -157782170

real    0m1.649s
user    0m3.236s
sys     0m0.028s
eduardo-argueta@vm:~/Desktop$ time ./mod5
Sum = 115887259

real    0m1.648s
user    0m3.248s
sys     0m0.008s
eduardo-argueta@vm:~/Desktop$ time ./mod5
Sum = -236367765

real    0m1.705s
user    0m3.205s
sys     0m0.008s
eduardo-argueta@vm:~/Desktop$
```

It can be seen that time has reduced than it was in modification#4. But it has still an issue. It is giving a wrong outcome i.e. it is giving incorrect sum. This is because, we have one global variable called *sum* which is being accessed by both the threads at the same time. This is called *race condition*.

Modification # 6

At last, to overcome our issue, we will be using pthread mutex locks. It will work like only 1 thread at a time can access that critical section i.e. in which we are accessing the *sum* global variable.

```
Open ▾    Save
                pthread_mutex_lock(&mutex);

                sum += offset;

                //End of critical section
                pthread_mutex_unlock(&mutex);
        }

        pthread_exit(NULL);
}

int main(void)
{
        //counting_function(1);

        //Spawn threads
        pthread_t id1;
        int offset1 = 1;
        pthread_create(&id1, NULL, counting_function, &offset1);

        pthread_t id2;
        int offset2 = -1;
        pthread_create(&id2, NULL, counting_function, &offset2);

        //Wait for threads
        pthread_join(id1, NULL);
        pthread_join(id2, NULL);

        printf("Sum = %lld\n", sum);
}
```



Now, we can see that we have gotten the correct outcome with 2 threads running parallel. There is again a drawback that it took a lot more time than previous modifications. The reason behind this is that we have are using 2 system calls of locking and unlocking the mutex in each iteration of for loop.