CS/IT 445 Software and Cybersecurity
LAB2

Name: Deewakar Goud

Roll No.: 202211016

Task 1: Comparison of different Ciphers

Introduction

Symmetric key block ciphers are important for keeping data safe. They encrypt and decrypt information quickly and securely. In this test, we'll compare four common encryption methods: DES, Triple DES (3DES), AES-128, and AES-256. We'll use Python and the PyCryptodome library to see how well they perform.

The whole experiment was built and run in **Python**. We used Python's standard tools for the main hash functions and added a few special, highly-optimized libraries for the others. The entire test was run from a single script to keep things consistent.

System Specs

- **Processor**: Intel Core i5-11300H @ 3.10GHz (4 cores, 8 threads)
- **RAM**: 8 GB DDR4
- **Operating System**: Windows 11 Home x64, Version 24H2
- **Python Version:** 3.12 (64-bit)
- **Library:** PyCryptodome 3.20.0

Ciphers Used

DES (Data Encryption Standard)

DES works by scrambling the data through a series of steps called a Feistel network. This process is repeated 16 times, or "rounds." Each round uses a different part of the key to mix up the data. Think of it like a deck of cards: after each round, the cards (or data bits) are shuffled in a new way based on a small piece of the key, making the original order harder to guess.

Triple DES (3DES/TDEA)

3DES gets its name because it performs the DES algorithm not once, but three times. The most common method is called encrypt-decrypt-encrypt (EDE). It works like this: first, the data is encrypted with key 1, then the result is decrypted with key 2, and finally, that result is encrypted again with key 3. This triple process makes the encryption much stronger, but also much slower. The decryption process simply reverses these steps.

AES (Advanced Encryption Standard)

AES is a more modern and complex algorithm. Instead of a Feistel network, it uses a substitution-permutation network. This means it performs four main operations in each round:

1. **SubBytes:** It replaces each byte of data with another byte from a special lookup table, like a secret codebook.
2. **ShiftRows:** It shifts the rows of the data block, scrambling the bytes to different positions.
3. **MixColumns:** It mathematically mixes the bytes in each column.
4. **AddRoundKey:** It combines the result with a part of the key for that specific round.

These steps are repeated for a number of rounds (10 for 128-bit keys, 12 for 192-bit keys, and 14 for 256-bit keys), making AES very secure and efficient. It's the standard used by governments and businesses worldwide.

Results

Sr. No.	Cipher	Key Size (bits)	Block Size (bits)	Avg Encrypt Time (s)	Avg Decrypt Time (s)	Priority
1	AES-128	128	128	0.002090	0.002322	1
2	AES-256	256	128	0.002282	0.002581	2
3	DES	56	64	0.014793	0.014389	3
4	Triple DES	168	64	0.040547	0.039857	4

Observations

- **AES (both 128 and 256 bit)** provided the fastest encryption and decryption times, confirming its status as the preferred standard for modern cryptography.
- **DES**, while faster than Triple DES, is much slower than AES and is not secure due to its short key size.
- **Triple DES** showed the highest computational cost, reflecting the extra rounds and keying material involved. This algorithm is rarely used today except for legacy compatibility.
- The marginal speed difference between AES-128 and AES-256 reflects AES's efficient design, although AES-256 has stronger security guarantees.

Source Code

```
import time import os from Crypto.Cipher
import DES, DES3, AES from
Crypto.Util.Padding import pad, unpad
from Crypto.Random import
get_random_bytes
 def
random_data(size_bytes=1024*1024):
    return os.urandom(size_bytes)
 def benchmark_cipher(name, cipher_constructor, key_generator,
block_size, data, mode):
    encrypt_times = []
decrypt_times = []     for
_ in range(10):
        key = key_generator()         iv =
get_random_bytes(block_size)         cipher =
cipher_constructor(key, mode, iv)         padded =
pad(data, block_size)
         start = time.perf_counter()         ct =
cipher.encrypt(padded)
encrypt_times.append(time.perf_counter() - start)
         cipher2 = cipher_constructor(key, mode, iv)
start = time.perf_counter()         pt =
unpad(cipher2.decrypt(ct), block_size)
decrypt_times.append(time.perf_counter() - start)
         assert pt == data  # sanity
check


    mean_encrypt = sum(encrypt_times) / len(encrypt_times)
mean_decrypt = sum(decrypt_times) / len(decrypt_times)
print(f"{name}: Encrypt Avg = {mean_encrypt:.6f} s, Decrypt Avg =
{mean_decrypt:.6f} s")



data = random_data() mode
= AES.MODE_CBC



benchmark_cipher(
    "DES",     DES.new,
lambda: get_random_bytes(8),
    8,
data,
DES.MODE_CBC
```

```
)
benchmark_cipher(
    "3DES",     DES3.new,     lambda:
DES3.adjust_key_parity(get_random_bytes(24)),
    8,
data,
    DES3.MODE_CBC
)
benchmark_cipher(
    "AES-128",     AES.new,
lambda: get_random_bytes(16),
    16,
data,
    AES.MODE_CBC
)
benchmark_cipher(
    "AES-256",     AES.new,
lambda: get_random_bytes(32),
    16,
data,
    AES.MODE_CBC
)
```

Task 2: Buffer Overflow Attack

Introduction

The main objective of this lab was to understand the concept of buffer overflow vulnerabilities. We practiced a real attack on a C program. Our goal was to fill a buffer with too much data so it would overflow. This overflow let us take control of the program and make it run a hidden function called secret() that the program normally doesn't use.

System Specs

- **Processor**: Intel Core i5-11300H @ 3.10GHz (4 cores, 8 threads)
- **RAM**: 8 GB DDR4
- **Operating System**: Windows 11 Home x64, Version 24H2 with WSL2 (Ubuntu 22.04 LTS)
- **GCC Version:** gcc (Ubuntu 11.4.0)
- **GDB Version:** GNU gdb (Ubuntu 15.0.50.20240403-git)

Determining the Offset

We found the offset by sending a long string of "A" characters to the program. By using a debugger like **GDB**, we observed that the program crashed when we sent 80 "A"s. We then looked at the state of the program and saw that the **saved return address** was overwritten specifically after the **72nd** "A" character. This confirmed that the offset, or the exact number of bytes needed to reach and overwrite the return address, was **72**.

Crafting the Payload

Instead of traditional **shellcode injection**, I constructed a payload using **Python**. The payload consisted of:

1. **Padding** – 72 As to fill the buffer and overwrite the saved RBP.
2. **Return Address** – the address of the secret() function in littleendian format.

The address of secret() was found in GDB:

Payload Construction

```
python3 -c 'import sys;
sys.stdout.buffer.write(b"A"*72 +
b"\x76\x11\x40\x00\x00\x00\x00\x00")'
```

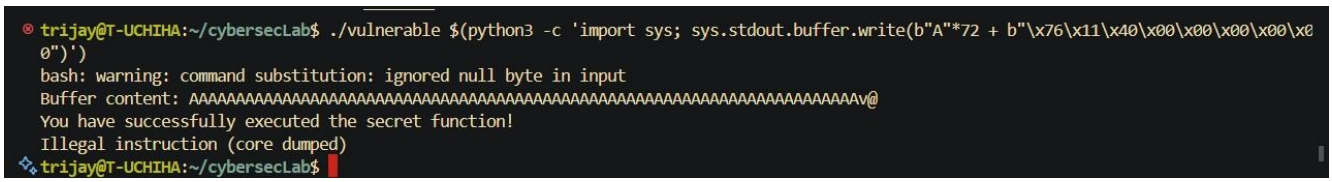
- `b"A"*72` → fills buffer + saved RBP.
- `\x76\x11\x40\x00\x00\x00\x00\x00` → overwrites RIP with address of secret().

Running the Exploit

Compiled with protections disabled:

```
gcc -fno-stack-protector -z execstack -no-pie
vulnerable.c -o vulnerable
```

Output



Explanation of Logic

- The buffer overflow overwrote the return address on the stack.
- Normally, the program would return to main after executing vulnerable_function.
- By overwriting RIP with the address of secret(), the program jumped directly into that function.
- This demonstrated how attackers can hijack execution flow using carefully crafted payloads.

Countermeasures

Buffer overflow vulnerabilities are dangerous and can lead to remote code execution. To mitigate such attacks, developers should adopt the following countermeasures:

1. **Avoid unsafe functions** – Replace strcpy, gets, and sprintf with safer alternatives (strncpy, fgets, snprintf).
2. **Enable stack protections** – Use compiler flags like -fstackprotector and modern mitigations like **stack canaries**.
3. **Control Flow Integrity (CFI)** – Ensures execution only follows valid control paths.
4. **Code reviews & static analysis** – Regular checks to detect unsafe memory usage.