LAB REPORT

5822UE Exercises: Security Insider Lab II - System and Application Security (Software-Sicherheit) - SS 2022

Part 5: Static Code Analysis, Dynamic Binary Instrumentation and Symbolic Execution

Group 2

Pratik Baishnav - 90760 (baish01@ads.uni-passau.de)

Walid Lombarkia - 107769 (lombar02@ads.uni-passau.de)

Date : 22nd June, 2022 - 6th July, 2022

Location: ITZ SR 002

Time : Wednesday (14:00 - 20:00)

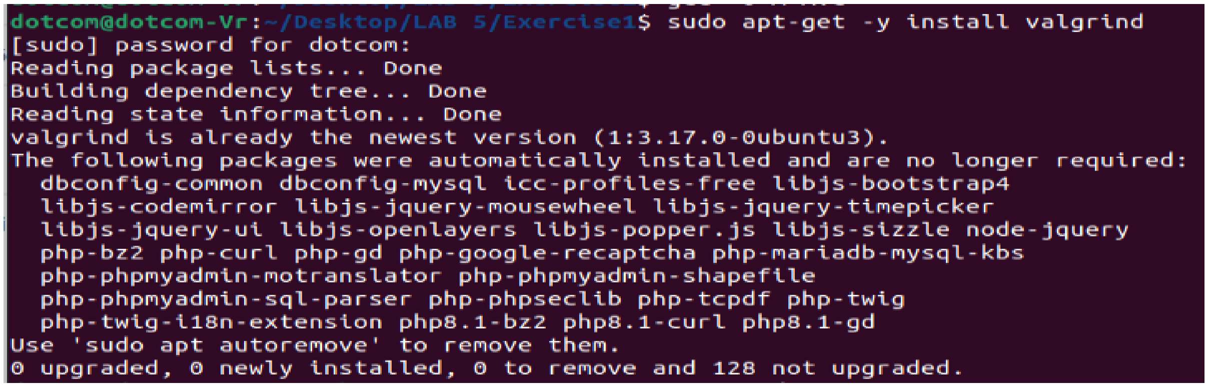
Organiser : Farnaz Mohammadi (Farnaz.Mohammadi@uni-passau.de)

Exercise 1 : Valgrind

1. **Compile the source code and then test the compiled executable with Valgrind.**

- We installed **Valgrind** using this command:

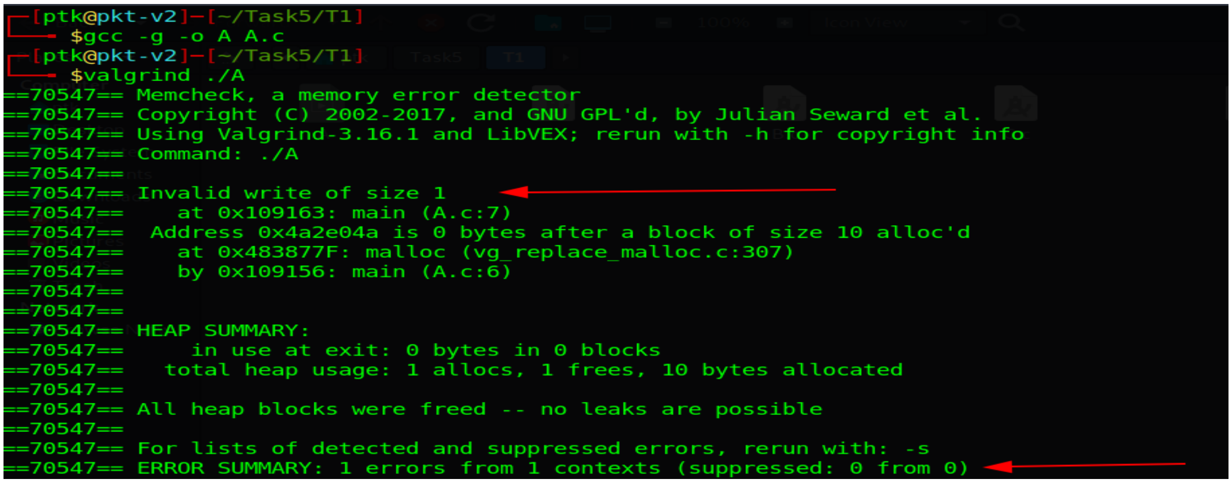
$ sudo apt-get -y install valgrind



```
dotcom@dotcom-Vr:~/Desktop/LAB 5/Exercise1$ sudo apt-get -y install valgrind
[sudo] password for dotcom:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
valgrind is already the newest version (1:3.17.0-0ubuntu3).
The following packages were automatically installed and are no longer required:
  dbconfig-common dbconfig-mysql icc-profiles-free libjs-bootstrap4
  libjs-codemirror libjs-jquery-mousewheel libjs-jquery-timepicker
  libjs-jquery-ui libjs-openlayers libjs-popper.js libjs-sizzle node-jquery
  php-bz2 php-curl php-gd php-google-recaptcha php-mariadb-mysql-kbs
  php-phpmyadmin-motranslator php-phpmyadmin-shapefile
  php-phpmyadmin-sql-parser php-phpseclib php-tcpdf php-twig
  php-twig-i18n-extension php8.1-bz2 php8.1-curl php8.1-gd
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 128 not upgraded.
```

- We compiled the source code using this command:
  - ➔ For Example A: **$ gcc -o A A.c**

- We test the compiled executable with Valgrind using the command: **$ valgrind ./A**

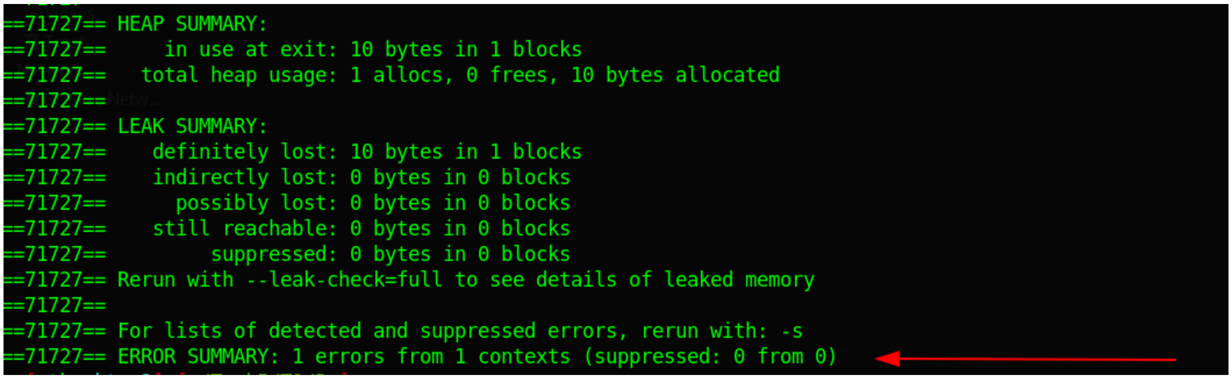


```
┌─[ptk@pkt-v2]─[~/Task5/T1]
└──╼ $gcc -g -o A A.c
┌─[ptk@pkt-v2]─[~/Task5/T1]
└──╼ $valgrind ./A
==70547== Memcheck, a memory error detector
==70547== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==70547== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==70547== Command: ./A
==70547==
==70547== Invalid write of size 1
==70547==    at 0x109163: main (A.c:7)
==70547==  Address 0x4a2e04a is 0 bytes after a block of size 10 alloc'd
==70547==    at 0x483877F: malloc (vg_replace_malloc.c:307)
==70547==    by 0x109156: main (A.c:6)
==70547==
==70547==
==70547== HEAP SUMMARY:
==70547==     in use at exit: 0 bytes in 0 blocks
==70547==   total heap usage: 1 allocs, 1 frees, 10 bytes allocated
==70547==
==70547== All heap blocks were freed -- no leaks are possible
==70547==
==70547== For lists of detected and suppressed errors, rerun with: -s
==70547== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

We did the same thing for the rest of the Examples.

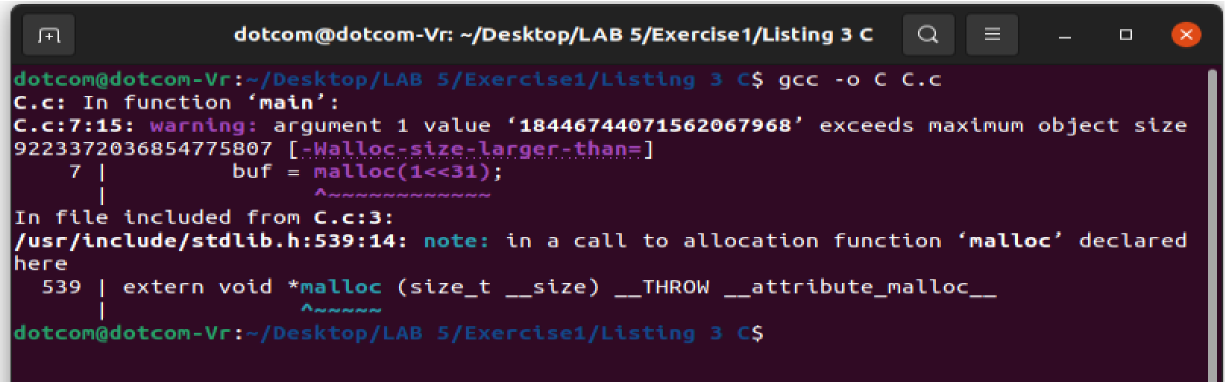
Part 5: Static Code Analysis, Dynamic Binary Instrumentation and Symbolic Execution

➔ Example B: **$ gcc -o B B.c**

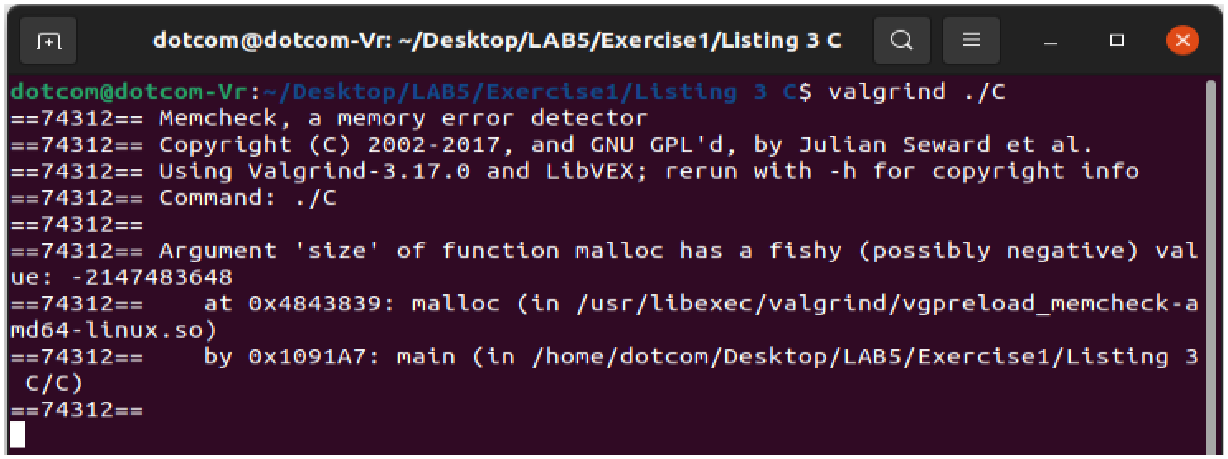


```
==71727== HEAP SUMMARY:
==71727==     in use at exit: 10 bytes in 1 blocks
==71727==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==71727==
==71727== LEAK SUMMARY:
==71727==    definitely lost: 10 bytes in 1 blocks
==71727==    indirectly lost: 0 bytes in 0 blocks
==71727==      possibly lost: 0 bytes in 0 blocks
==71727==    still reachable: 0 bytes in 0 blocks
==71727==         suppressed: 0 bytes in 0 blocks
==71727== Rerun with --leak-check=full to see details of leaked memory
==71727==
==71727== For lists of detected and suppressed errors, rerun with: -s
==71727== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

➔ Example C: **$ gcc -o C C.c**

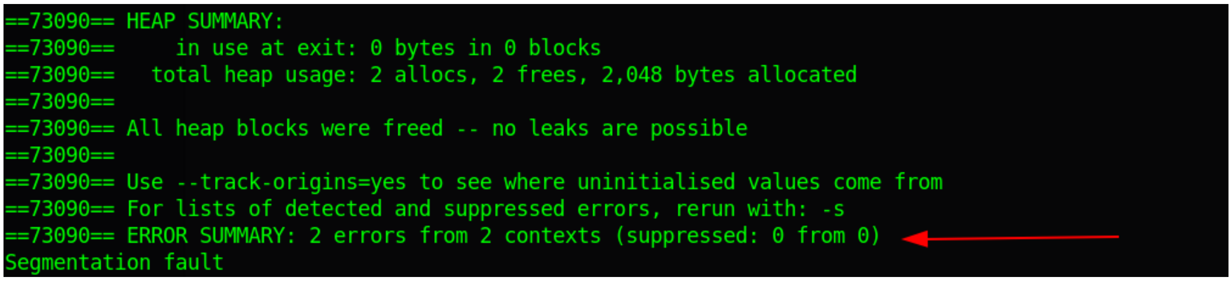


```
dotcom@dotcom-Vr:~/Desktop/LAB 5/Exercise1/Listing 3 C$ gcc -o C C.c
C.c: In function ‘main’:
C.c:7:15: warning: argument 1 value ‘18446744071562067968’ exceeds maximum object size
9223372036854775807 [-Walloc-size-larger-than=]
    7 |           buf = malloc(1<<31);
      |                 ^~~~~~~~~~~~~
In file included from C.c:3:
/usr/include/stdlib.h:539:14: note: in a call to allocation function ‘malloc’ declared
here
  539 | extern void *malloc (size_t __size) __THROW __attribute_malloc__
      |              ^~~~~~
dotcom@dotcom-Vr:~/Desktop/LAB 5/Exercise1/Listing 3 C$
```



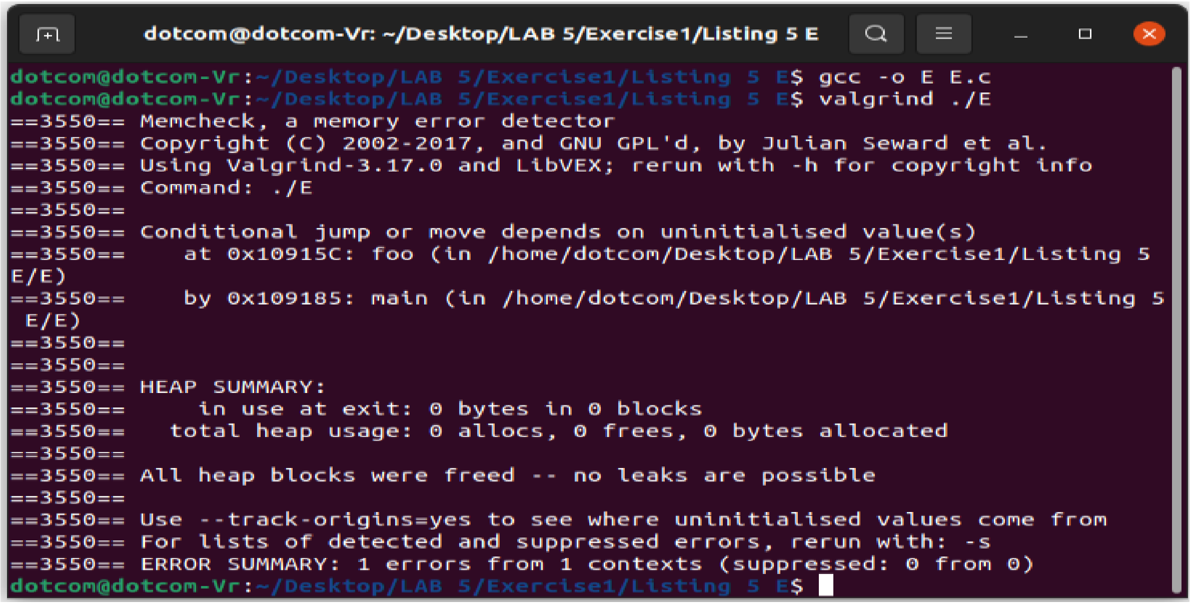
```
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise1/Listing 3 C$ valgrind ./C
==74312== Memcheck, a memory error detector
==74312== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==74312== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info
==74312== Command: ./C
==74312==
==74312== Argument 'size' of function malloc has a fishy (possibly negative) val
ue: -2147483648
==74312==    at 0x4843839: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-a
md64-linux.so)
==74312==    by 0x1091A7: main (in /home/dotcom/Desktop/LAB5/Exercise1/Listing 3
 C/C)
==74312==
```

➔ Example D: **$ gcc -o D D.c**



```
==73090== HEAP SUMMARY:
==73090==     in use at exit: 0 bytes in 0 blocks
==73090==   total heap usage: 2 allocs, 2 frees, 2,048 bytes allocated
==73090==
==73090== All heap blocks were freed -- no leaks are possible
==73090==
==73090== Use --track-origins=yes to see where uninitialised values come from
==73090== For lists of detected and suppressed errors, rerun with: -s
==73090== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
Segmentation fault
```

➔ Example E: **$ gcc -o E E.c**



```
dotcom@dotcom-Vr:~/Desktop/LAB 5/Exercise1/Listing 5 E$ gcc -o E E.c
dotcom@dotcom-Vr:~/Desktop/LAB 5/Exercise1/Listing 5 E$ valgrind ./E
==3550== Memcheck, a memory error detector
==3550== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3550== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info
==3550== Command: ./E
==3550==
==3550== Conditional jump or move depends on uninitialised value(s)
==3550==    at 0x10915C: foo (in /home/dotcom/Desktop/LAB 5/Exercise1/Listing 5
E/E)
==3550==    by 0x109185: main (in /home/dotcom/Desktop/LAB 5/Exercise1/Listing 5
 E/E)
==3550==
==3550==
==3550== HEAP SUMMARY:
==3550==     in use at exit: 0 bytes in 0 blocks
==3550==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==3550==
==3550== All heap blocks were freed -- no leaks are possible
==3550==
==3550== Use --track-origins=yes to see where uninitialised values come from
==3550== For lists of detected and suppressed errors, rerun with: -s
==3550== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
dotcom@dotcom-Vr:~/Desktop/LAB 5/Exercise1/Listing 5 E$ 
```

2. **What errors you detected? What is the name of that category of errors?**

- **Example A:**

  ➔ The array size was 10 but it was trying to access x[10] which was out of that array size.

  ➔ **Category of the error: Array index out of bounds.**

- **Example B:**

  ➔ The array size was 10 but it was trying to access x[10] which was out of that array size.

➔ **Category of the error: Array index out of bounds + dealocation.**

❖ **Example C:**

➔ 1<<31   0x80000000   -2147483648

➔ **Category of the error: No real memory allocation error, array index out of bounds.**

❖ **Example D:**

➔ Missing operator in scanf scanf("%d", y);

➔ **Category of the error: Absence of reference operator (&) in scanf.**

❖ **Example E:**

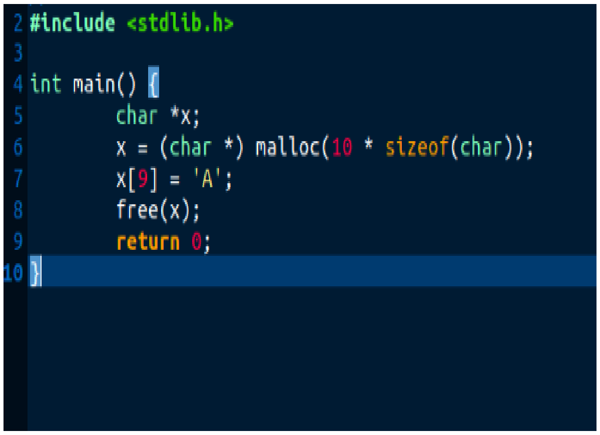
➔ X is not initialiated

➔ **Category of the error: Passing uninitialized variable or undefined variable.**

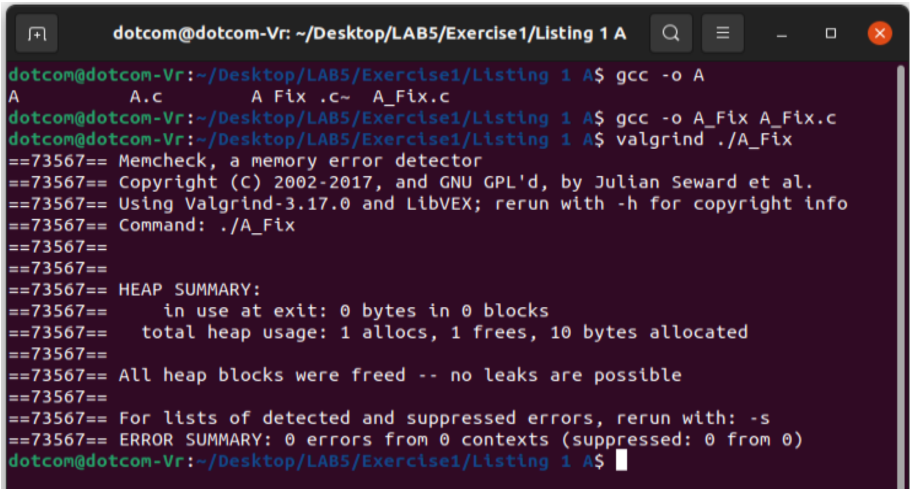
3. Fix the errors in the source code, recompile and retest it again with Valgrind. Make sure that it does not generate errors this time.

❖ **Example A:**

➔ **To fix the problem we need to reduce the size of the variable x.**



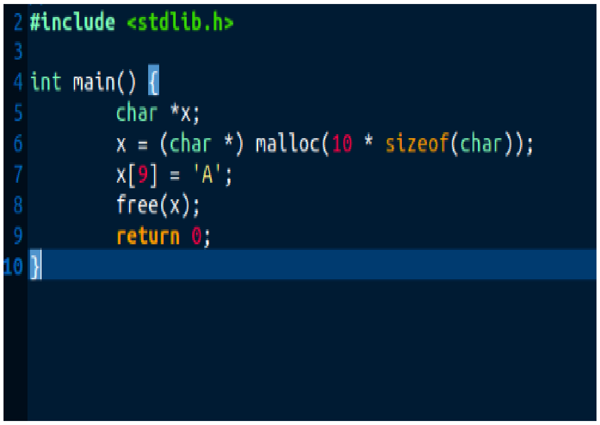
```c
#include <stdlib.h>

int main() {
        char *x;
        x = (char *) malloc(10 * sizeof(char));
        x[9] = 'A';
        free(x);
        return 0;
}
```



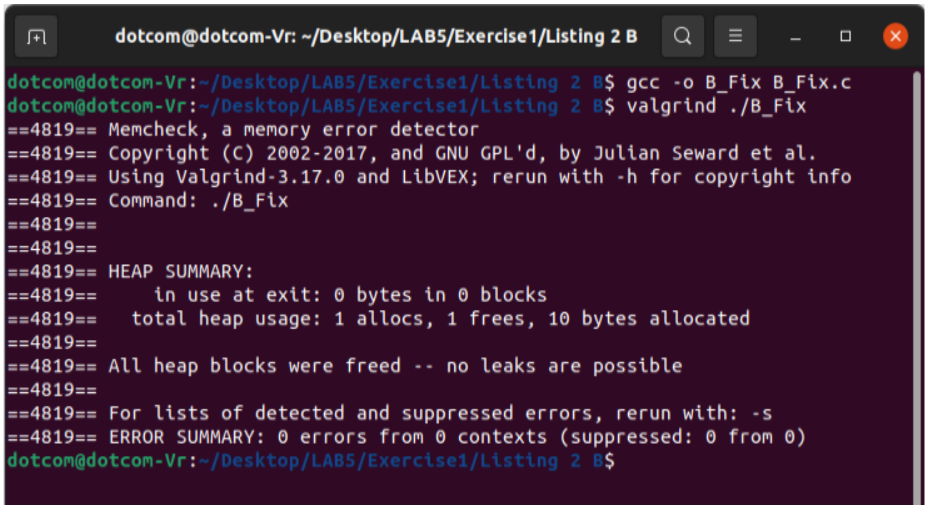
```
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise1/Listing 1 A$ gcc -o A
A          A.c        A Fix .c~  A_Fix.c
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise1/Listing 1 A$ gcc -o A_Fix A_Fix.c
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise1/Listing 1 A$ valgrind ./A_Fix
==73567== Memcheck, a memory error detector
==73567== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==73567== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info
==73567== Command: ./A_Fix
==73567==
==73567==
==73567== HEAP SUMMARY:
==73567==     in use at exit: 0 bytes in 0 blocks
==73567==   total heap usage: 1 allocs, 1 frees, 10 bytes allocated
==73567==
==73567== All heap blocks were freed -- no leaks are possible
==73567==
==73567== For lists of detected and suppressed errors, rerun with: -s
==73567== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise1/Listing 1 A$ 
```

- ❖ **Example B:**

  - ➔ **To fix the problem we need to reduce the size of the variable x.**
  - ➔ **Also, if you're returning allocated memory, when we call the function, it has to free the memory.**

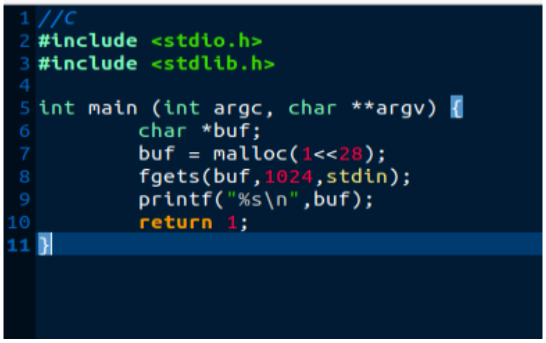


```c
#include <stdlib.h>

int main() {
        char *x;
        x = (char *) malloc(10 * sizeof(char));
        x[9] = 'A';
        free(x);
        return 0;
}
```

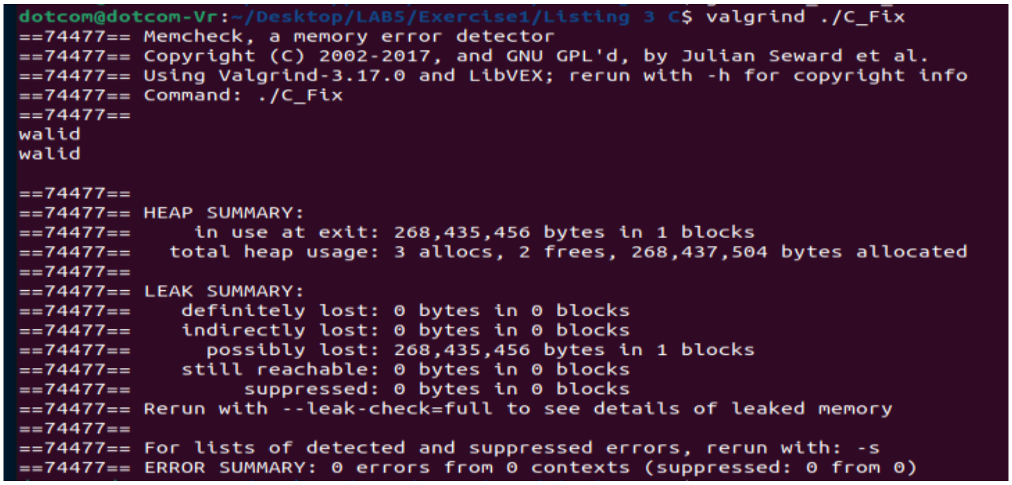


```
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise1/Listing 2 B$ gcc -o B_Fix B_Fix.c
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise1/Listing 2 B$ valgrind ./B_Fix
==4819== Memcheck, a memory error detector
==4819== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4819== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info
==4819== Command: ./B_Fix
==4819==
==4819==
==4819== HEAP SUMMARY:
==4819==     in use at exit: 0 bytes in 0 blocks
==4819==   total heap usage: 1 allocs, 1 frees, 10 bytes allocated
==4819==
==4819== All heap blocks were freed -- no leaks are possible
==4819==
==4819== For lists of detected and suppressed errors, rerun with: -s
==4819== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise1/Listing 2 B$
```

- ❖ **Example C:**



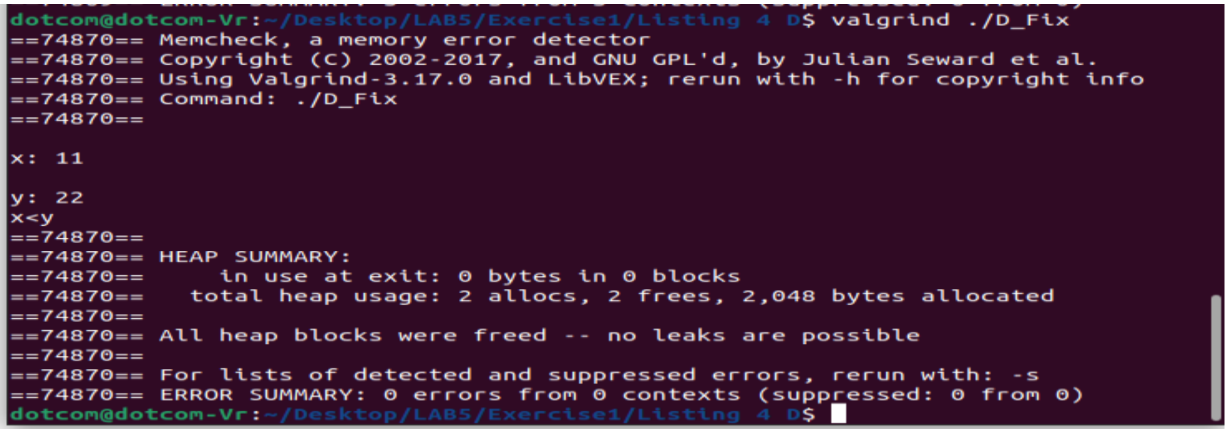
```c
//C
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char **argv) {
        char *buf;
        buf = malloc(1<<28);
        fgets(buf,1024,stdin);
        printf("%s\n",buf);
        return 1;
}
```



```
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise1/Listing 3 C$ valgrind ./C_Fix
==74477== Memcheck, a memory error detector
==74477== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==74477== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info
==74477== Command: ./C_Fix
==74477==
walid
walid

==74477==
==74477== HEAP SUMMARY:
==74477==     in use at exit: 268,435,456 bytes in 1 blocks
==74477==   total heap usage: 3 allocs, 2 frees, 268,437,504 bytes allocated
==74477==
==74477== LEAK SUMMARY:
==74477==    definitely lost: 0 bytes in 0 blocks
==74477==    indirectly lost: 0 bytes in 0 blocks
==74477==      possibly lost: 268,435,456 bytes in 1 blocks
==74477==    still reachable: 0 bytes in 0 blocks
==74477==         suppressed: 0 bytes in 0 blocks
==74477== Rerun with --leak-check=full to see details of leaked memory
==74477==
==74477== For lists of detected and suppressed errors, rerun with: -s
==74477== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

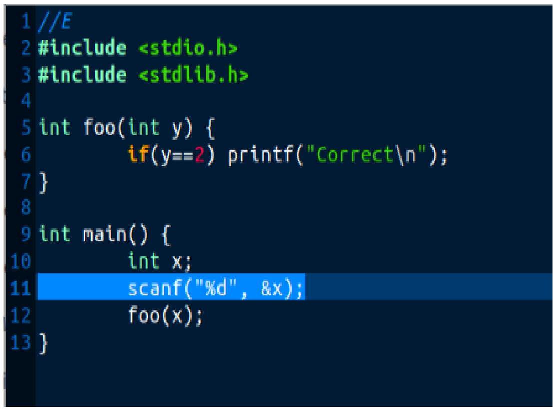
- ❖ **Example D:**

  - ➔ To fix the problem we added **"&"** in front of "y" in the **scanf("%d",&y); .**

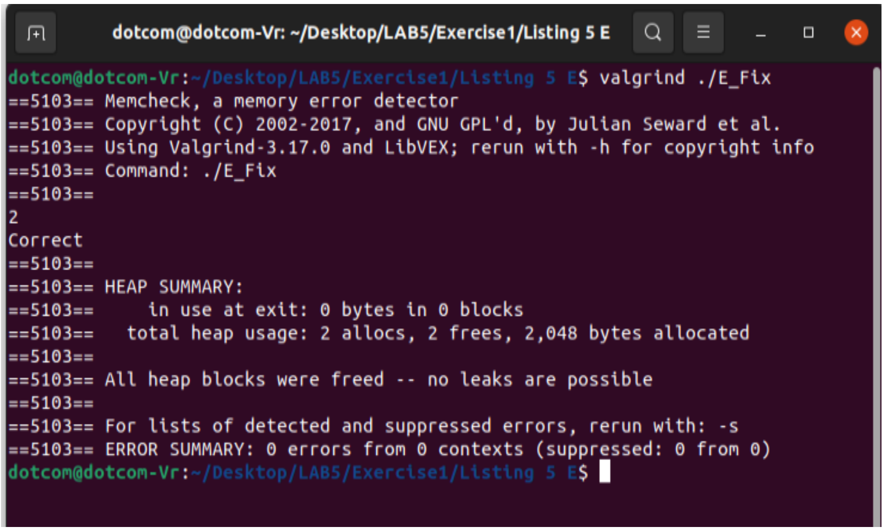


```
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise1/Listing 4 D$ valgrind ./D_Fix
==74870== Memcheck, a memory error detector
==74870== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==74870== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info
==74870== Command: ./D_Fix
==74870==

x: 11

y: 22
x<y
==74870==
==74870== HEAP SUMMARY:
==74870==     in use at exit: 0 bytes in 0 blocks
==74870==   total heap usage: 2 allocs, 2 frees, 2,048 bytes allocated
==74870==
==74870== All heap blocks were freed -- no leaks are possible
==74870==
==74870== For lists of detected and suppressed errors, rerun with: -s
==74870== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise1/Listing 4 D$
```

❖ **Example E:**



```c
//E
#include <stdio.h>
#include <stdlib.h>

int foo(int y) {
        if(y==2) printf("Correct\n");
}

int main() {
        int x;
        scanf("%d", &x);
        foo(x);
}
```



```
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise1/Listing 5 E$ valgrind ./E_Fix
==5103== Memcheck, a memory error detector
==5103== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5103== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info
==5103== Command: ./E_Fix
==5103==
2
Correct
==5103==
==5103== HEAP SUMMARY:
==5103==     in use at exit: 0 bytes in 0 blocks
==5103==   total heap usage: 2 allocs, 2 frees, 2,048 bytes allocated
==5103==
==5103== All heap blocks were freed -- no leaks are possible
==5103==
==5103== For lists of detected and suppressed errors, rerun with: -s
==5103== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise1/Listing 5 E$
```

Exercise 2: KLEE

1. **Include the file ”klee.h” in you source code.**

   ➔ In the source code we add **: #include <klee/klee.h>**

2. **Choose which variable(s) should be symbolic by using the function” klee_make_symbolic” and adjust the code.**

   ➔ In the source code we add: **klee_make_symbolic(&usrInput, sizeof(int), "usrInput");** and adjusted the code.

3. **Compile the source code into LLVM bitcode using for example the command llvm-gcc --emit-llvm.**

First of all, we need to **install KLEE** so for this the steps are as follows:

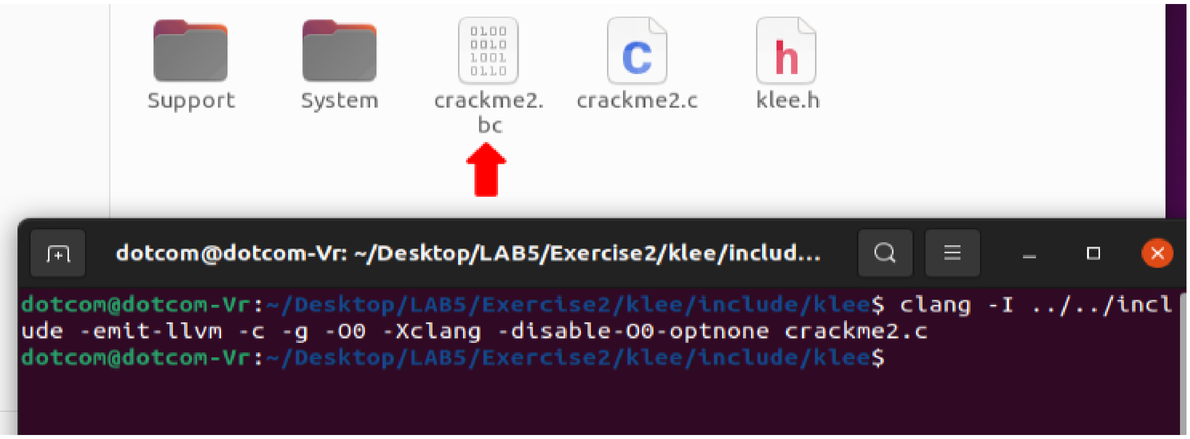
- ➔ **Docker pull klee/klee**
- ➔ **$ git clone https://github.com/klee/klee.git**
- ➔ **$ cd klee**
- ➔ **$ docker build -t klee/klee .**

To run KLEE:

- ➔ Then we mounted a host directory to klee docker instance, Now things will reflect on both sides: **docker run -ti -v /home/ptk/Task5/T2:/home/klee/src --name=ptk --ulimit='stack=-1:-1' klee/klee** .
- ➔ To restart the same container: **sudo docker start -ai ptk**

To compile :

- ➔ **clang -I ~/klee_src/include/ -emit-llvm -c -g crackme2.c**
- ➔ we get **crackme2.c** .

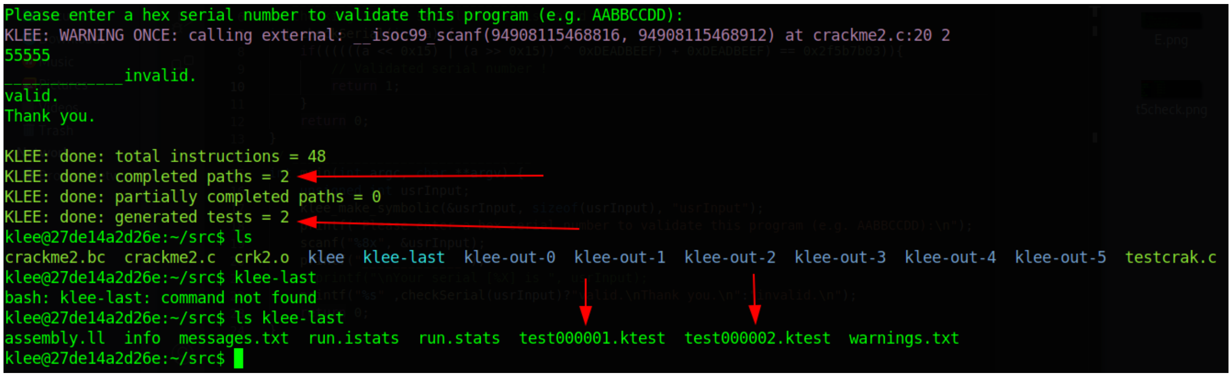


```
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise2/klee/include/klee$ clang -I ../../incl
ude -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone crackme2.c
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise2/klee/include/klee$
```

4. **Run KLEE on the resulting LLVM bitcode to generate the test cases.**

We have crackme2.bc file which we run using,

- ➔ **Klee crackme2.bc**

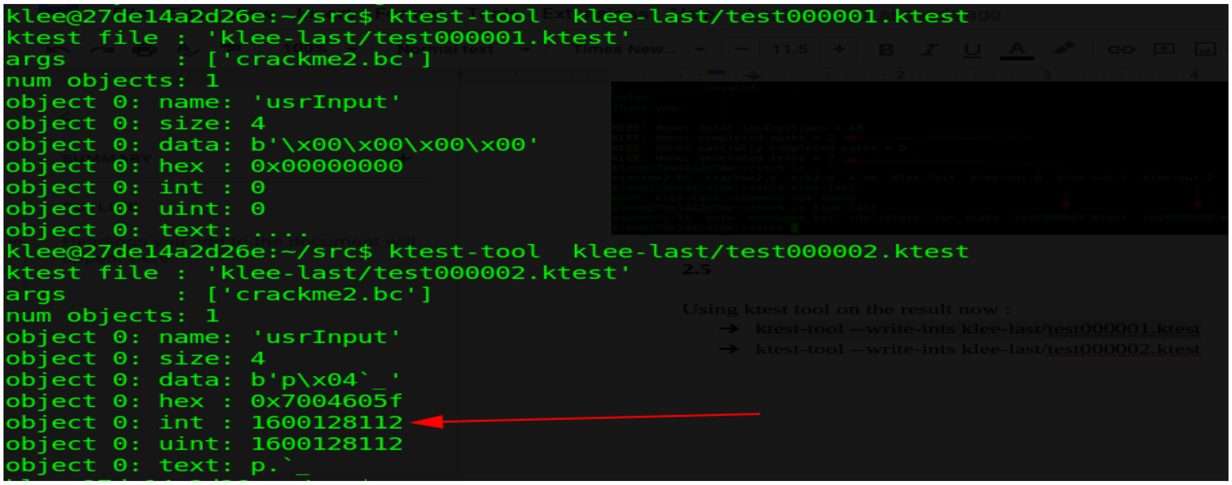


```
Please enter a hex serial number to validate this program (e.g. AABBCCDD):
KLEE: WARNING ONCE: calling external: __isoc99_scanf(94908115468816, 94908115468912) at crackme2.c:20 2
55555
_____________invalid.
valid.
Thank you.

KLEE: done: total instructions = 48
KLEE: done: completed paths = 2
KLEE: done: partially completed paths = 0
KLEE: done: generated tests = 2
klee@27de14a2d26e:~/src$ ls
crackme2.bc  crackme2.c  crk2.o  klee  klee-last  klee-out-0  klee-out-1  klee-out-2  klee-out-3  klee-out-4  klee-out-5  testcrak.c
klee@27de14a2d26e:~/src$ klee-last
bash: klee-last: command not found
klee@27de14a2d26e:~/src$ ls klee-last
assembly.ll  info  messages.txt  run.istats  run.stats  test000001.ktest  test000002.ktest  warnings.txt
klee@27de14a2d26e:~/src$
```

5. **Use the command ktest-tool on the result cases and decide which one solves the problem.**

Using ktest tool on the result now :

- ➔ **ktest-tool --write-ints klee-last/test000001.ktest**
- ➔ **ktest-tool --write-ints klee-last/test000002.ktest**

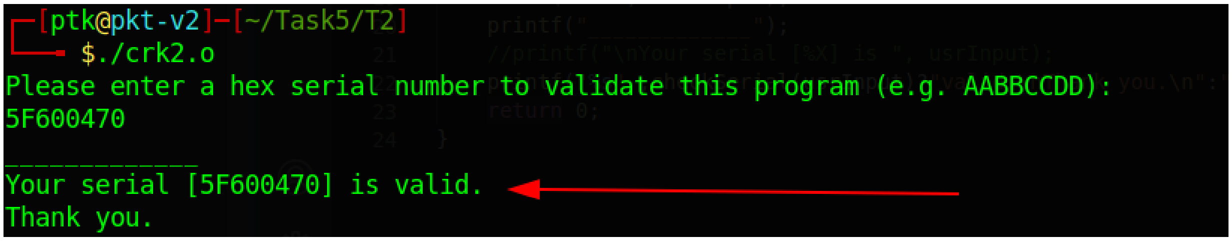


```
klee@27de14a2d26e:~/src$ ktest-tool  klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['crackme2.bc']
num objects: 1
object 0: name: 'usrInput'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x00'
object 0: hex : 0x00000000
object 0: int : 0
object 0: uint: 0
object 0: text: ....
klee@27de14a2d26e:~/src$ ktest-tool  klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['crackme2.bc']
num objects: 1
object 0: name: 'usrInput'
object 0: size: 4
object 0: data: b'p\x04`_'
object 0: hex : 0x7004605f
object 0: int : 1600128112
object 0: uint: 1600128112
object 0: text: p.`_
```

- ➔ We got integer **1600128112** as a result.

6. **Is this result valid?**

- ➔ We first converted the int into hex value and tested with crk2.o and the result is valid.
- ➔ **1600128112: 5F600470** and when we use **5F600470**, this key is valid.



```
┌─[ptk@pkt-v2]─[~/Task5/T2]
└──╼ $./crk2.o
Please enter a hex serial number to validate this program (e.g. AABBCCDD):
5F600470

_____________
Your serial [5F600470] is valid.
Thank you.
```

7. **Find some (at least three) valid keys and show that they are valid keys by entering them into the original program. Show how you achieved that.**

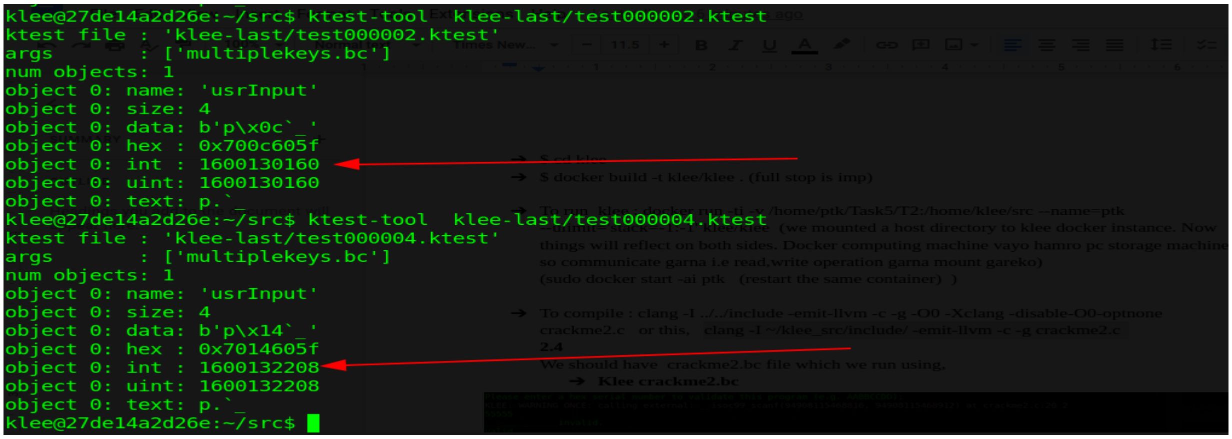
➔ For this, we modified the code a little bit by creating more test cases.

Modified code:

```
int checkSerial(int a) {
if((((((a << 0x15) | (a >> 0x15)) ^ 0xDEADBEEF) +0xDEADBEEF) == 0x2f5b7b03)){}

if(a!=1600128112 && (((((a << 0x15) | (a >> 0x15)) ^0xDEADBEEF) + 0xDEADBEEF) ==
0x2f5b7b03)){}

if(a!=1600130160 && a!=1600128112 && (((((a << 0x15) | (a>> 0x15)) ^ 0xDEADBEEF) +
0xDEADBEEF) == 0x2f5b7b03)){}
return 0;
}
```

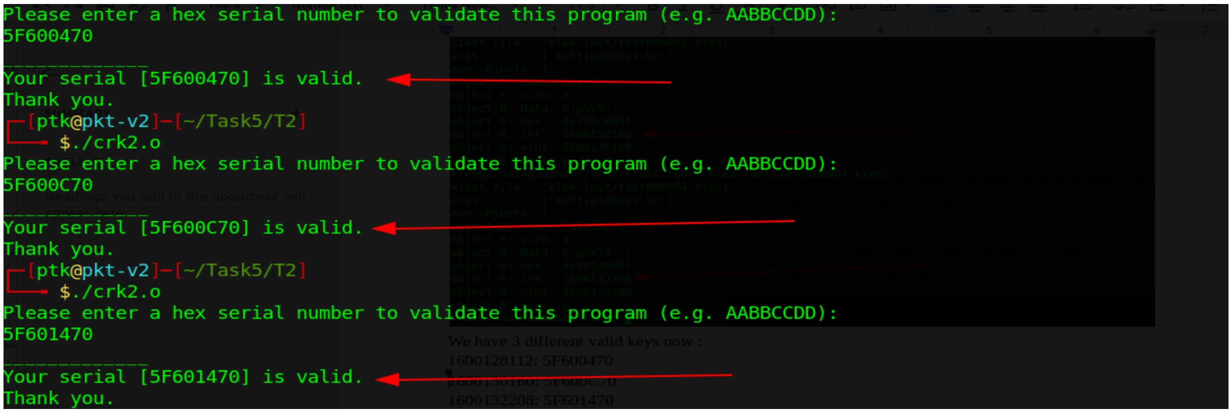
➔ We got 2 more different valid keys and in total, we have 3 different keys.

1600128112: 5F600470
1600130160: 5F600C70
1600132208: 5F601470



```
klee@27de14a2d26e:~/src$ ktest-tool  klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['multiplekeys.bc']
num objects: 1
object 0: name: 'usrInput'
object 0: size: 4
object 0: data: b'p\x0c`_'
object 0: hex : 0x700c605f
object 0: int : 1600130160
object 0: uint: 1600130160
object 0: text: p.`_
klee@27de14a2d26e:~/src$ ktest-tool  klee-last/test000004.ktest
ktest file : 'klee-last/test000004.ktest'
args       : ['multiplekeys.bc']
num objects: 1
object 0: name: 'usrInput'
object 0: size: 4
object 0: data: b'p\x14`_'
object 0: hex : 0x7014605f
object 0: int : 1600132208
object 0: uint: 1600132208
object 0: text: p.`_
klee@27de14a2d26e:~/src$
```

➔ Again we converted these integers to Hexa decimal and tried these to check the validity in the crackme2.o .



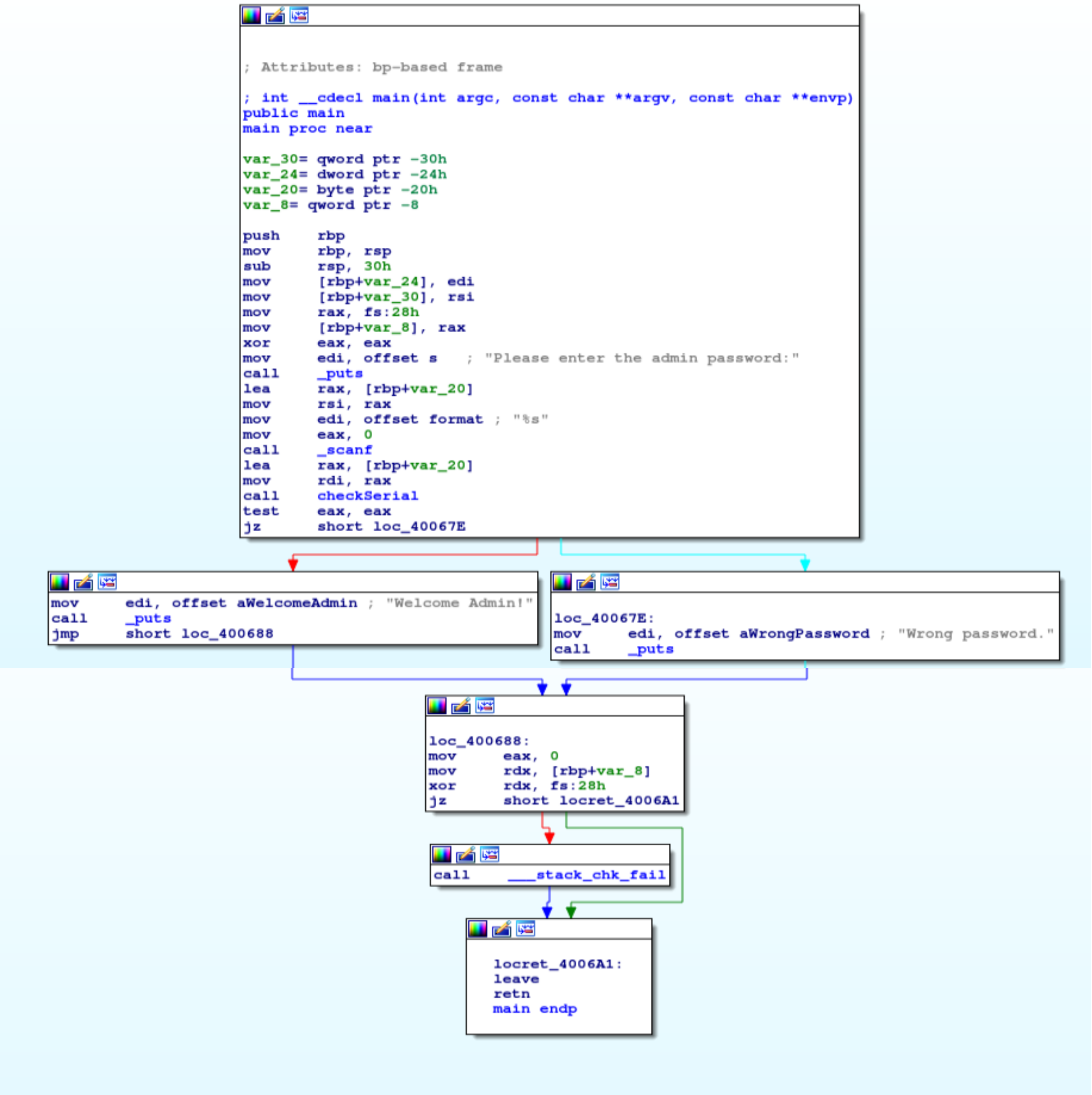
```
Please enter a hex serial number to validate this program (e.g. AABBCCDD):
5F600470
_____________
Your serial [5F600470] is valid.
Thank you.
┌─[ptk@pkt-v2]─[~/Task5/T2]
└──╼ $./crk2.o
Please enter a hex serial number to validate this program (e.g. AABBCCDD):
5F600C70
_____________
Your serial [5F600C70] is valid.
Thank you.
┌─[ptk@pkt-v2]─[~/Task5/T2]
└──╼ $./crk2.o
Please enter a hex serial number to validate this program (e.g. AABBCCDD):
5F601470
_____________
Your serial [5F601470] is valid.
Thank you.
```

Exercise 3: angr

1. Using IDA or any other tool, show the Control Flow Graph (CFG) of the program.

➔ Download IDA from the official website and install it

➔ To run go to **/opt/ida/idafree-7.7** then **./ida64** to run the program.



```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_30= qword ptr -30h
var_24= dword ptr -24h
var_20= byte ptr -20h
var_8= qword ptr -8

push    rbp
mov     rbp, rsp
sub     rsp, 30h
mov     [rbp+var_24], edi
mov     [rbp+var_30], rsi
mov     rax, fs:28h
mov     [rbp+var_8], rax
xor     eax, eax
mov     edi, offset s   ; "Please enter the admin password:"
call    _puts
lea     rax, [rbp+var_20]
mov     rsi, rax
mov     edi, offset format ; "%s"
mov     eax, 0
call    _scanf
lea     rax, [rbp+var_20]
mov     rdi, rax
call    checkSerial
test    eax, eax
jz      short loc_40067E
```

```
mov     edi, offset aWelcomeAdmin ; "Welcome Admin!"
call    _puts
jmp     short loc_400688
```

```
loc_40067E:
mov     edi, offset aWrongPassword ; "Wrong password."
call    _puts
```

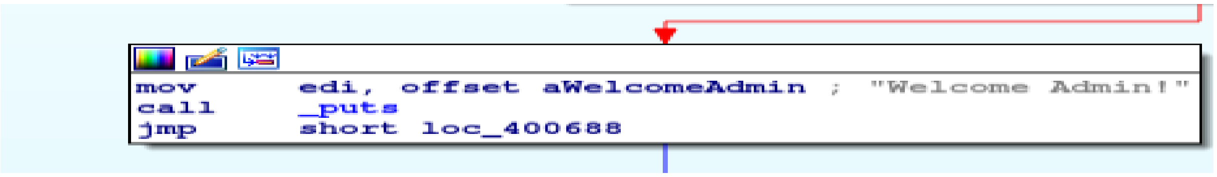
```
loc_400688:
mov     eax, 0
mov     rdx, [rbp+var_8]
xor     rdx, fs:28h
jz      short locret_4006A1
```

```
call    ___stack_chk_fail
```

```
locret_4006A1:
leave
retn
main endp
```

2. Indicate which branch corresponds to the valid serial.

There is **<checkSerial>** function which checks for conditional statement. If the condition evaluates to true then it takes the path corresponds to valid serial and prints **Welcome Admin!** where an invalid serial leads to **Wrong Password.**



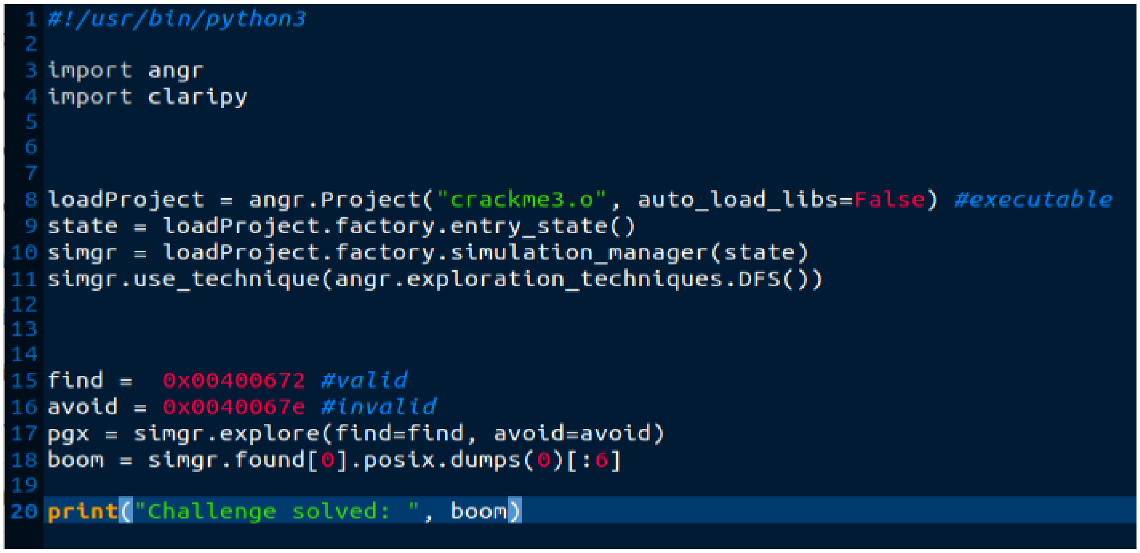
```
mov     edi, offset aWelcomeAdmin ; "Welcome Admin!"
call    _puts
jmp     short loc_400688
```

3.Write a python script with using angr to solve the task and give a valid serial.

To Install angr:

- ➔ sudo apt-get install python3-dev libffi-dev build-essential
- ➔ python3 -m pip install --user virtualenv
- ➔ python3 -m venv ang
- ➔ source ang/bin/activate
- ➔ pip install angr
- ➔ After this simply run **python3 test.py (**we get the code**)**

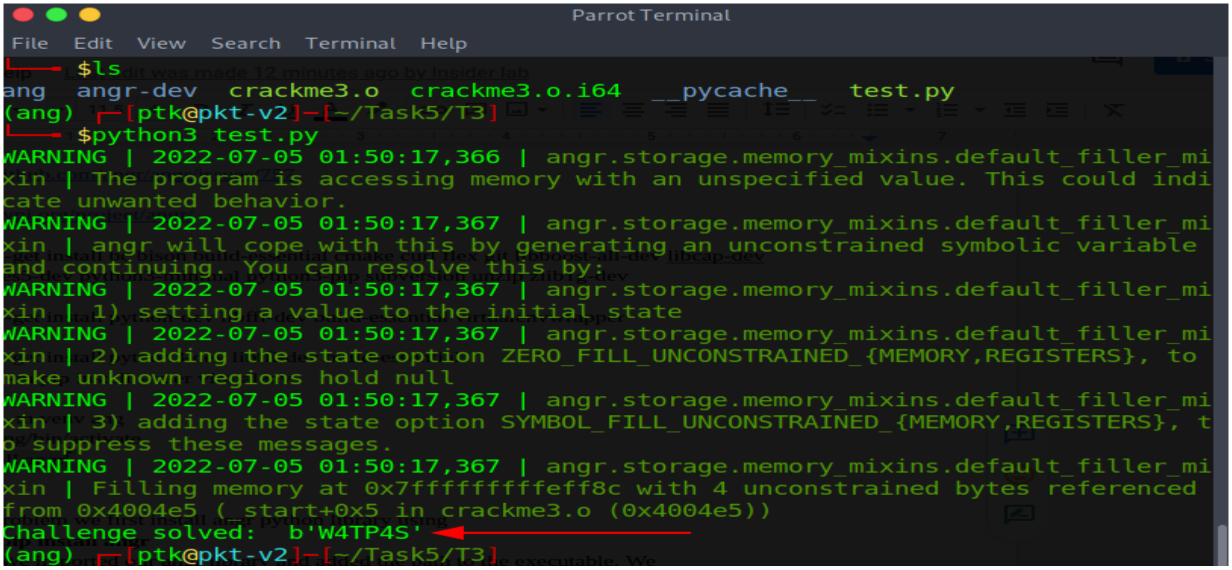
Our Python Script:



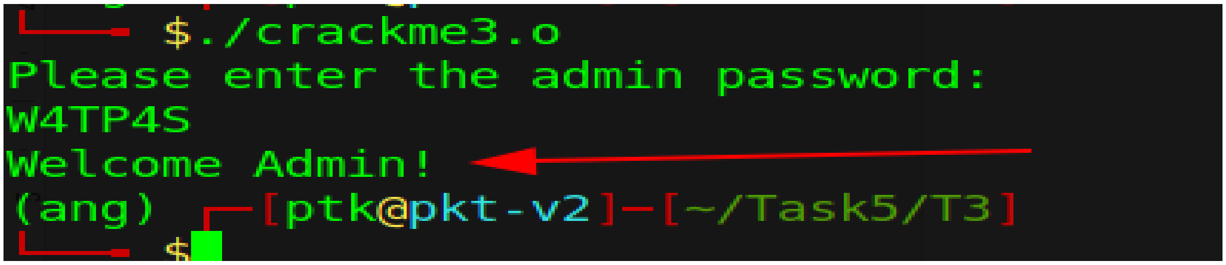
```
#!/usr/bin/python3

import angr
import claripy



loadProject = angr.Project("crackme3.o", auto_load_libs=False) #executable
state = loadProject.factory.entry_state()
simgr = loadProject.factory.simulation_manager(state)
simgr.use_technique(angr.exploration_techniques.DFS())



find =  0x00400672 #valid
avoid = 0x0040067e #invalid
pgx = simgr.explore(find=find, avoid=avoid)
boom = simgr.found[0].posix.dumps(0)[:6]

print("Challenge solved: ", boom)
```

To execute our Python Script:

➔ **source ang/bin/activate** we get: **W4TP4S** and we run it in the original program crackme3.o .



```
└──── $ls
ang  angr-dev  crackme3.o  crackme3.o.i64  __pycache__  test.py
(ang) ┌─[ptk@pkt-v2]─[~/Task5/T3]
└──── $python3 test.py
WARNING | 2022-07-05 01:50:17,366 | angr.storage.memory_mixins.default_filler_mi
xin | The program is accessing memory with an unspecified value. This could indi
cate unwanted behavior.
WARNING | 2022-07-05 01:50:17,367 | angr.storage.memory_mixins.default_filler_mi
xin | angr will cope with this by generating an unconstrained symbolic variable
and continuing. You can resolve this by:
WARNING | 2022-07-05 01:50:17,367 | angr.storage.memory_mixins.default_filler_mi
xin | 1) setting a value to the initial state
WARNING | 2022-07-05 01:50:17,367 | angr.storage.memory_mixins.default_filler_mi
xin | 2) adding the state option ZERO_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to
make unknown regions hold null
WARNING | 2022-07-05 01:50:17,367 | angr.storage.memory_mixins.default_filler_mi
xin | 3) adding the state option SYMBOL_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, t
o suppress these messages.
WARNING | 2022-07-05 01:50:17,367 | angr.storage.memory_mixins.default_filler_mi
xin | Filling memory at 0x7fffffffffeff8c with 4 unconstrained bytes referenced
from 0x4004e5 (_start+0x5 in crackme3.o (0x4004e5))
Challenge solved:  b'W4TP4S'
(ang) ┌─[ptk@pkt-v2]─[~/Task5/T3]
```



```
└──── $./crackme3.o
Please enter the admin password:
W4TP4S
Welcome Admin!
(ang) ┌─[ptk@pkt-v2]─[~/Task5/T3]
└──── $
```

Exercise 4: American Fuzzy Lop (afl)

1. **Some of the testers remarked that is crashes sometimes. One tester just remembered that the input that caused the crash was 4 lowercase chars (this info is just to speed your fuzzing).**

   American fuzzy lop is designed to check unexpected user inputs to simulate crashes or behaviors and expose vulnerabilities in some code bases. First of all, we install afl to do this in the terminal we run:

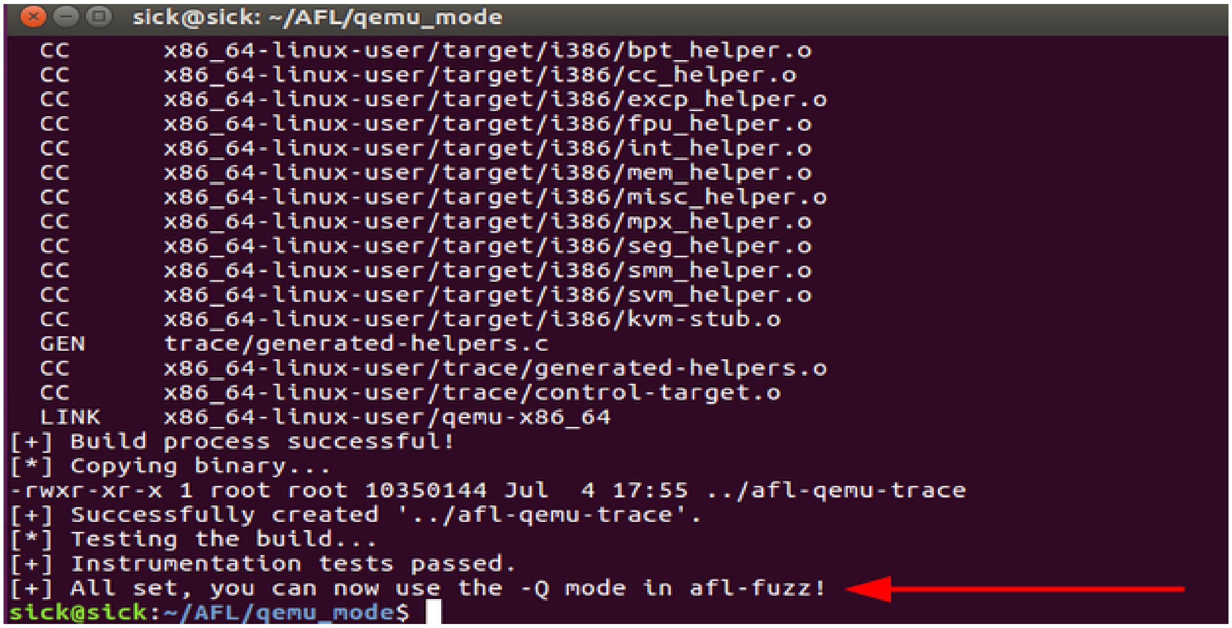
   ➔ git clone https://github.com/google/AFL.git
   ➔ cd afl

- ➔ Make
- ➔ Install all dependencies which we don't have(shown in the error.)
- ➔ Then, we edit our ~/**.bashrc** to point the command to the directory we extract.
- ➔ **alias afl-fuzz="$HOME/path-to-afl/afl-fuzz"**
- ➔ **alias afl-tmin="$HOME/path-to-afl/afl-tmin"**
- ➔ **alias afl-showmap="$HOME/path-to-afl/afl-showmap"**
- ➔ Restart your shell: **exec $SHELL**

2. **Compile afl with QEMU support. Why we need this?**

   Setting up QEMU mode :

   - ➔ **Cd qemu_mode**
   - ➔ **Sudo ./build_qemu_support.sh (**which will run afl in qemu mode)
   - ➔ QEMU mode is required because we don't have access to the source code to compile the program with afl gcc.



```
  CC      x86_64-linux-user/target/i386/bpt_helper.o
  CC      x86_64-linux-user/target/i386/cc_helper.o
  CC      x86_64-linux-user/target/i386/excp_helper.o
  CC      x86_64-linux-user/target/i386/fpu_helper.o
  CC      x86_64-linux-user/target/i386/int_helper.o
  CC      x86_64-linux-user/target/i386/mem_helper.o
  CC      x86_64-linux-user/target/i386/misc_helper.o
  CC      x86_64-linux-user/target/i386/mpx_helper.o
  CC      x86_64-linux-user/target/i386/seg_helper.o
  CC      x86_64-linux-user/target/i386/smm_helper.o
  CC      x86_64-linux-user/target/i386/svm_helper.o
  CC      x86_64-linux-user/target/i386/kvm-stub.o
  GEN     trace/generated-helpers.c
  CC      x86_64-linux-user/trace/generated-helpers.o
  CC      x86_64-linux-user/trace/control-target.o
  LINK    x86_64-linux-user/qemu-x86_64
[+] Build process successful!
[*] Copying binary...
-rwxr-xr-x 1 root root 10350144 Jul  4 17:55 ../afl-qemu-trace
[+] Successfully created '../afl-qemu-trace'.
[*] Testing the build...
[+] Instrumentation tests passed.
[+] All set, you can now use the -Q mode in afl-fuzz!
sick@sick:~/AFL/qemu_mode$
```

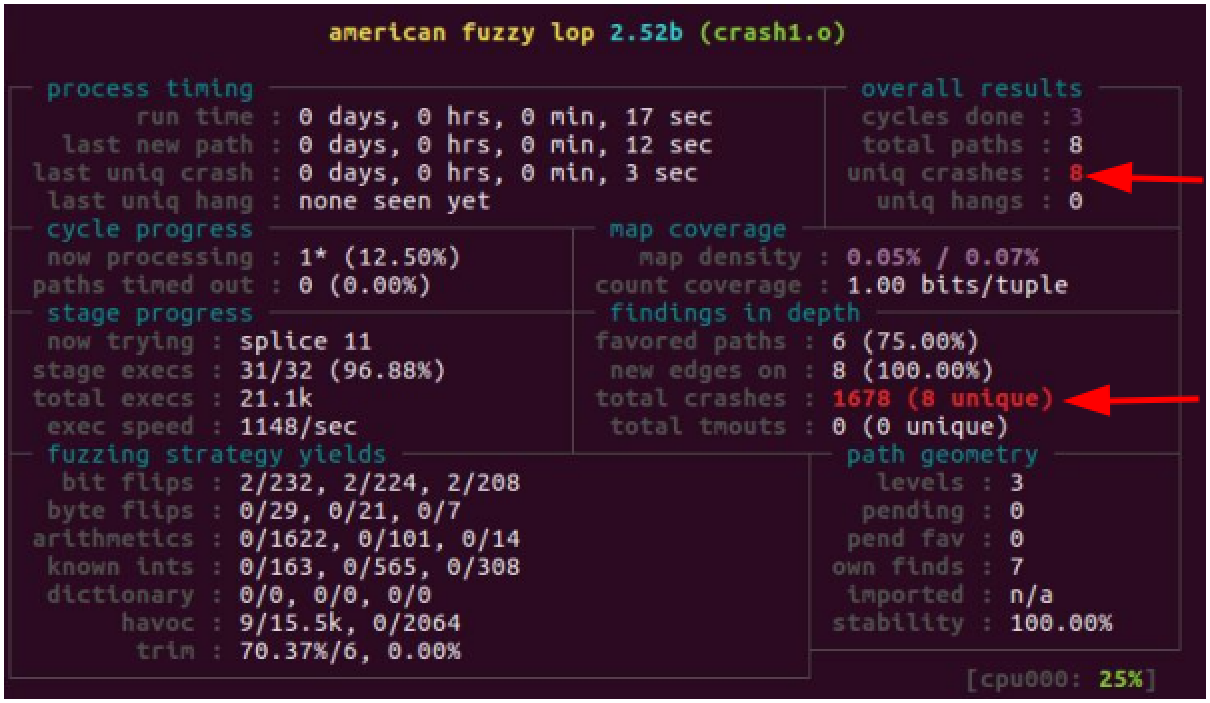
3. **Launch your fuzzing using afl-fuzz with adjusted parameters to find the crash case. Remember: The requested crash case is readable.**

   Now, that the QEMU mode is ready we can run **afl** with the following command :

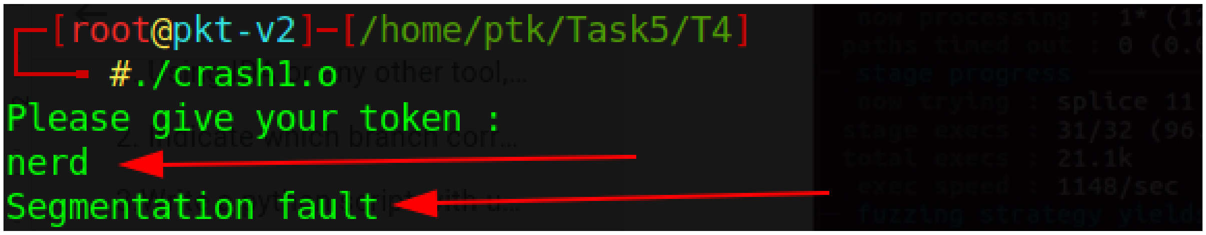
   - ➔ **afl-fuzz -i testcases/others/text/ -o output -Q ./crash1.o**

   where, -i to specify the input test case, -o to specify the output directory, -Q to run in QEMU mode.

   - ➔ We get the 4 small case string "**nerd".**
   - ➔ We run the executable file and put the string value we got to crash the program.



```
                   american fuzzy lop 2.52b (crash1.o)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 17 sec      │  cycles done : 3      │
│   last new path : 0 days, 0 hrs, 0 min, 12 sec      │  total paths : 8      │
│ last uniq crash : 0 days, 0 hrs, 0 min, 3 sec       │ uniq crashes : 8      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 1* (12.50%)       │    map density : 0.05% / 0.07%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : splice 11             │ favored paths : 6 (75.00%)             │
│ stage execs : 31/32 (96.88%)        │  new edges on : 8 (100.00%)            │
│ total execs : 21.1k                 │ total crashes : 1678 (8 unique)        │
│  exec speed : 1148/sec              │  total tmouts : 0 (0 unique)           │
├─ fuzzing strategy yields ───────────┴───────────┬─ path geometry ────────────┤
│   bit flips : 2/232, 2/224, 2/208               │    levels : 3              │
│  byte flips : 0/29, 0/21, 0/7                   │   pending : 0              │
│ arithmetics : 0/1622, 0/101, 0/14               │  pend fav : 0              │
│  known ints : 0/163, 0/565, 0/308               │ own finds : 7              │
│  dictionary : 0/0, 0/0, 0/0                     │  imported : n/a            │
│       havoc : 9/15.5k, 0/2064                   │ stability : 100.00%        │
│        trim : 70.37%/6, 0.00%                   ├────────────────────────────┘
└─────────────────────────────────────────────────┘          [cpu000: 25%]
```

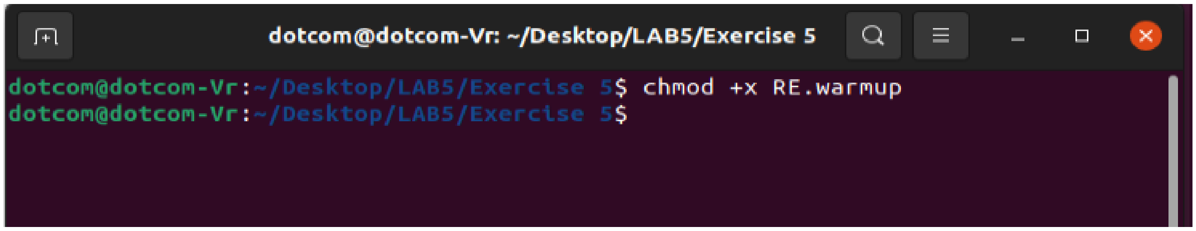


```
┌─[root@pkt-v2]─[/home/ptk/Task5/T4]
└──╼ #./crash1.o
Please give your token :
nerd
Segmentation fault
```

Exercise 5: Generic Reverse Engineering and Malware Analysis

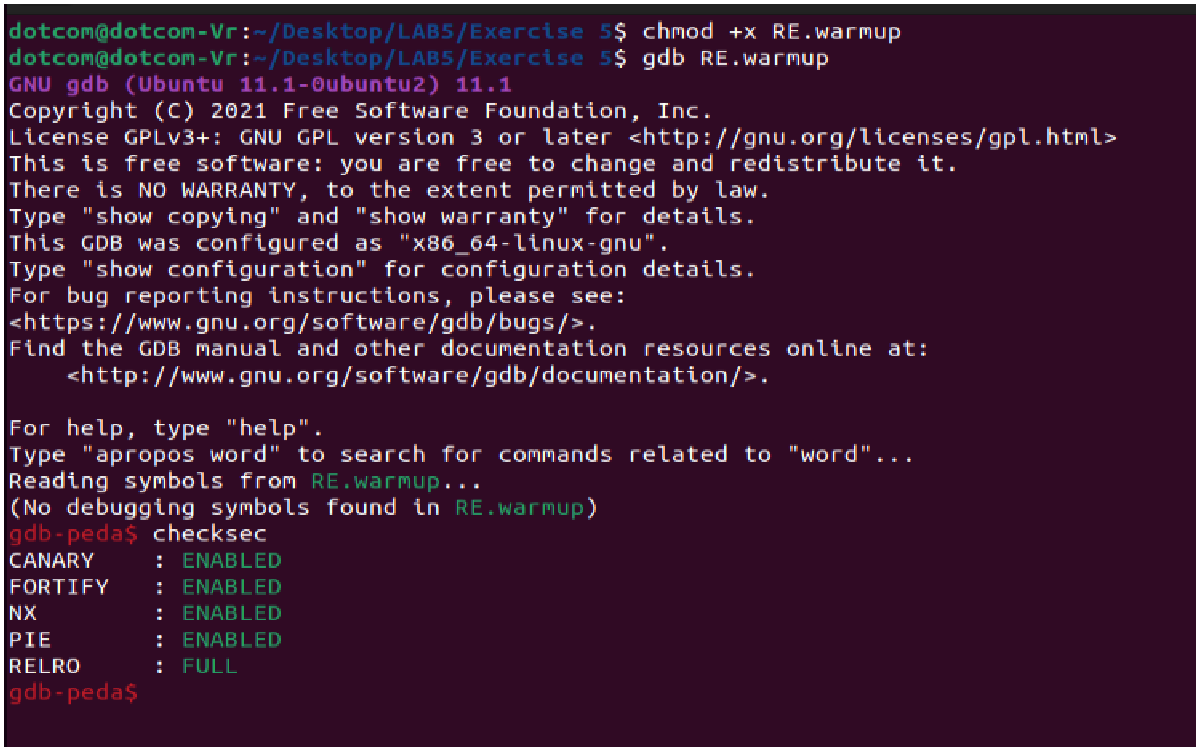
1. Check for implemented security features. What did you find?

First, we make the code executable with this command: **$chmod +x RE.warmup**



```
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise 5$ chmod +x RE.warmup
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise 5$
```

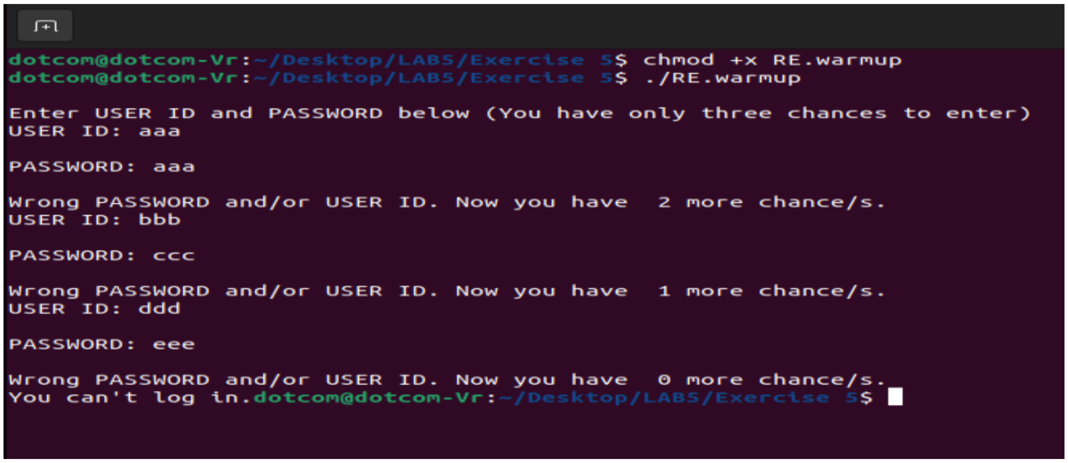
Here we checked the security features using **gdb** with this command: **$ checksec**



```
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise 5$ chmod +x RE.warmup
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise 5$ gdb RE.warmup
GNU gdb (Ubuntu 11.1-0ubuntu2) 11.1
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from RE.warmup...
(No debugging symbols found in RE.warmup)
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : ENABLED
NX        : ENABLED
PIE       : ENABLED
RELRO     : FULL
gdb-peda$
```

2. **The Program is asking for credentials to access. +**
3. **Retrieve the correct credentials and use them to login. And explain how you break it?**

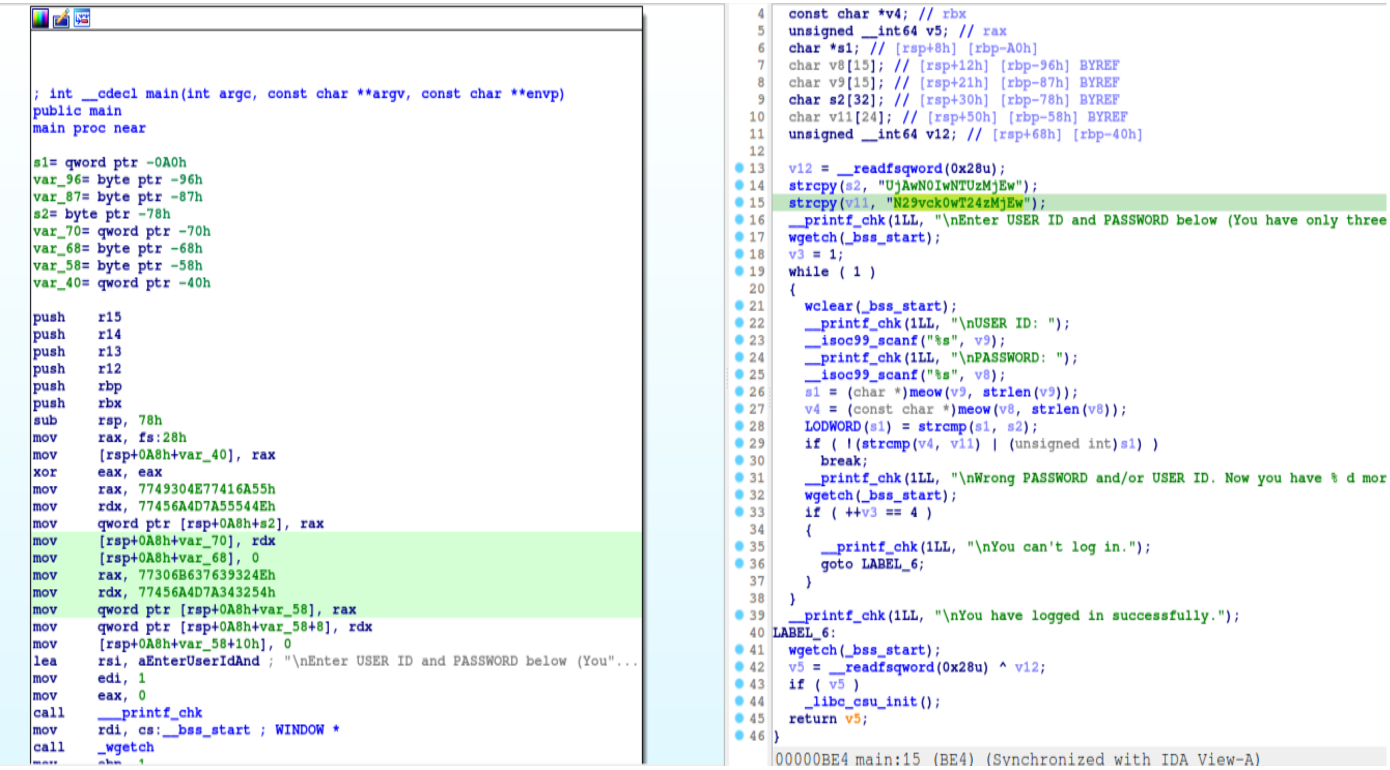
We run it and it asks us to enter the USER ID and Password (it gives us 3 chances to enter the correct ones)



```
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise 5$ chmod +x RE.warmup
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise 5$ ./RE.warmup

Enter USER ID and PASSWORD below (You have only three chances to enter)
USER ID: aaa

PASSWORD: aaa

Wrong PASSWORD and/or USER ID. Now you have  2 more chance/s.
USER ID: bbb

PASSWORD: ccc

Wrong PASSWORD and/or USER ID. Now you have  1 more chance/s.
USER ID: ddd

PASSWORD: eee

Wrong PASSWORD and/or USER ID. Now you have  0 more chance/s.
You can't log in.dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise 5$ 
```

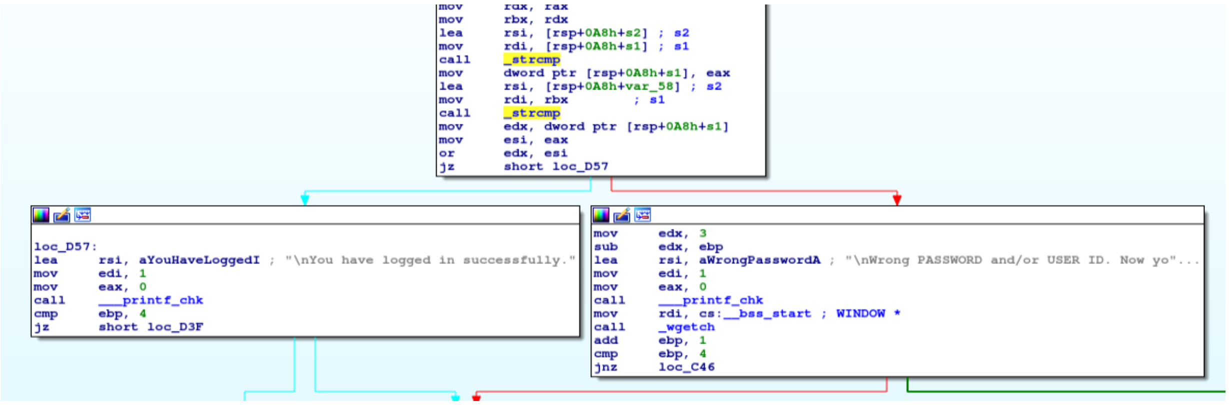
Now we run IDA and upload the executable file to IDA tool and we generate a pseuducode

- ➔ We look for strcmp() function that compares the UserID and password provided by the user and the ones stored in the register
- ➔ After that, we can find from our pseudocode strcpy functions, the variables.



```
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

s1= qword ptr -0A0h
var_96= byte ptr -96h
var_87= byte ptr -87h
s2= byte ptr -78h
var_70= qword ptr -70h
var_68= byte ptr -68h
var_58= byte ptr -58h
var_40= qword ptr -40h

push    r15
push    r14
push    r13
push    r12
push    rbp
push    rbx
sub     rsp, 78h
mov     rax, fs:28h
mov     [rsp+0A8h+var_40], rax
xor     eax, eax
mov     rax, 7749304E77416A55h
mov     rdx, 77456A4D7A55544Eh
mov     qword ptr [rsp+0A8h+s2], rax
mov     [rsp+0A8h+var_70], rdx
mov     [rsp+0A8h+var_68], 0
mov     rax, 77306B637639324Eh
mov     rdx, 77456A4D7A343254h
mov     qword ptr [rsp+0A8h+var_58], rax
mov     qword ptr [rsp+0A8h+var_58+8], rdx
mov     [rsp+0A8h+var_58+10h], 0
lea     rsi, aEnterUserIdAnd ; "\nEnter USER ID and PASSWORD below (You"...
mov     edi, 1
mov     eax, 0
call    ___printf_chk
mov     rdi, cs:__bss_start ; WINDOW *
call    _wgetch
```

```
  const char *v4; // rbx
  unsigned __int64 v5; // rax
  char *s1; // [rsp+8h] [rbp-A0h]
  char v8[15]; // [rsp+12h] [rbp-96h] BYREF
  char v9[15]; // [rsp+21h] [rbp-87h] BYREF
  char s2[32]; // [rsp+30h] [rbp-78h] BYREF
  char v11[24]; // [rsp+50h] [rbp-58h] BYREF
  unsigned __int64 v12; // [rsp+68h] [rbp-40h]

  v12 = __readfsqword(0x28u);
  strcpy(s2, "UjAwN0IwNTUzMjEw");
  strcpy(v11, "N29vck0wT24zMjEw");
  __printf_chk(1LL, "\nEnter USER ID and PASSWORD below (You have only three
  wgetch(_bss_start);
  v3 = 1;
  while ( 1 )
  {
    wclear(_bss_start);
    __printf_chk(1LL, "\nUSER ID: ");
    __isoc99_scanf("%s", v9);
    __printf_chk(1LL, "\nPASSWORD: ");
    __isoc99_scanf("%s", v8);
    s1 = (char *)meow(v9, strlen(v9));
    v4 = (const char *)meow(v8, strlen(v8));
    LODWORD(s1) = strcmp(s1, s2);
    if ( !(strcmp(v4, v11) | (unsigned int)s1) )
      break;
    __printf_chk(1LL, "\nWrong PASSWORD and/or USER ID. Now you have % d mor
    wgetch(_bss_start);
    if ( ++v3 == 4 )
    {
      __printf_chk(1LL, "\nYou can't log in.");
      goto LABEL_6;
    }
  }
  __printf_chk(1LL, "\nYou have logged in successfully.");
LABEL_6:
  wgetch(_bss_start);
  v5 = __readfsqword(0x28u) ^ v12;
  if ( v5 )
    _libc_csu_init();
  return v5;
}
00000BE4 main:15 (BE4) (Synchronized with IDA View-A)
```

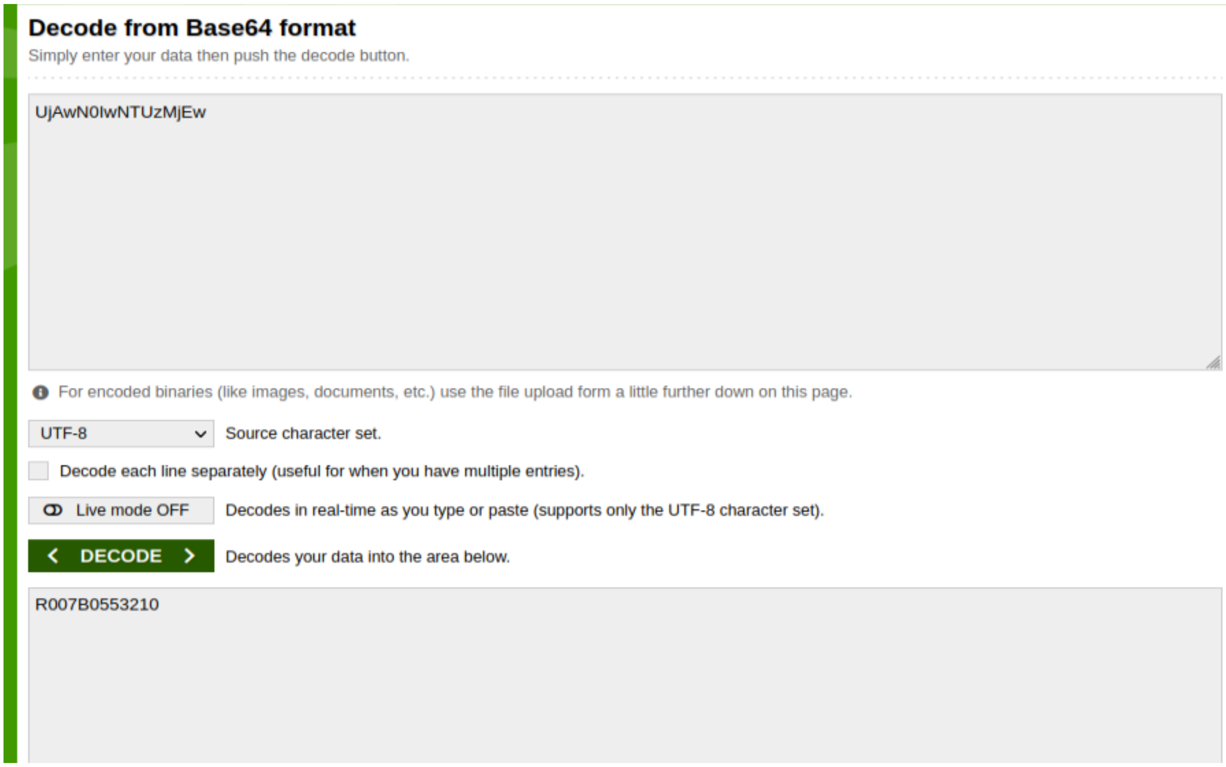


```
mov     rdx, rax
mov     rbx, rdx
lea     rsi, [rsp+0A8h+s2] ; s2
mov     rdi, [rsp+0A8h+s1] ; s1
call    _strcmp
mov     dword ptr [rsp+0A8h+s1], eax
lea     rsi, [rsp+0A8h+var_58] ; s2
mov     rdi, rbx        ; s1
call    _strcmp
mov     edx, dword ptr [rsp+0A8h+s1]
mov     esi, eax
or      edx, esi
jz      short loc_D57
```

```
loc_D57:
lea     rsi, aYouHaveLoggedI ; "\nYou have logged in successfully."
mov     edi, 1
mov     eax, 0
call    ___printf_chk
cmp     ebp, 4
jz      short loc_D3F
```

```
mov     edx, 3
sub     edx, ebp
lea     rsi, aWrongPasswordA ; "\nWrong PASSWORD and/or USER ID. Now yo"...
mov     edi, 1
mov     eax, 0
call    ___printf_chk
mov     rdi, cs:__bss_start ; WINDOW *
call    _wgetch
add     ebp, 1
cmp     ebp, 4
jnz     loc_C46
```

all we need to do now is to convert it using a base64 decoder:



Decode from Base64 format

Simply enter your data then push the decode button.

```
UjAwN0lwNTUzMjEw
```

For encoded binaries (like images, documents, etc.) use the file upload form a little further down on this page.

UTF-8 — Source character set.

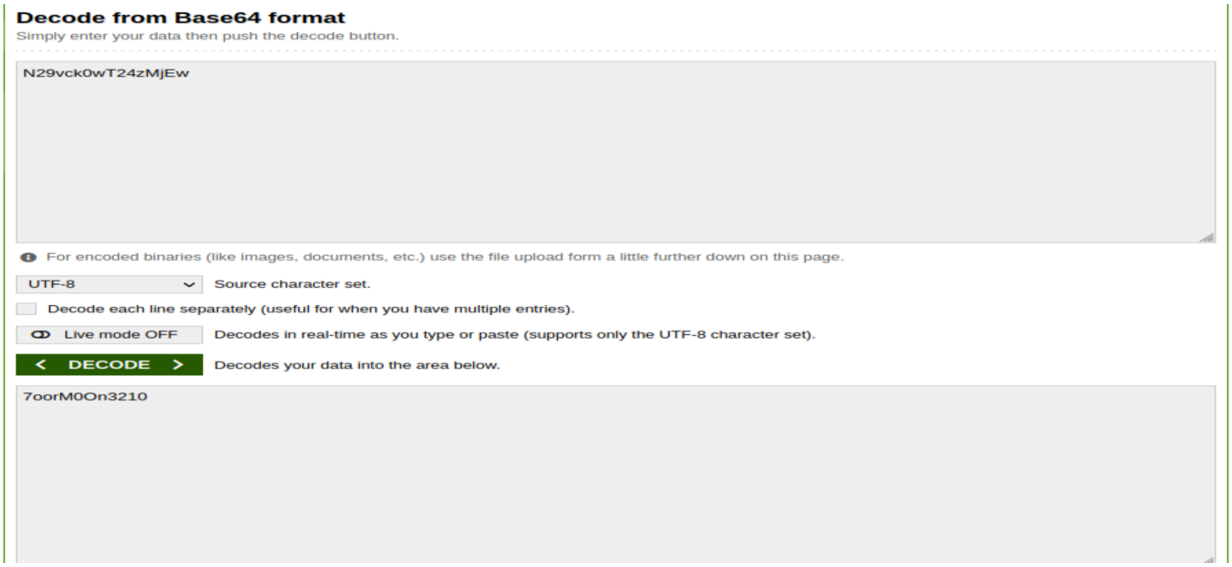
□ Decode each line separately (useful for when you have multiple entries).

Live mode OFF — Decodes in real-time as you type or paste (supports only the UTF-8 character set).

< DECODE > — Decodes your data into the area below.

```
R007B0553210
```

- ➔ **R007B0553210** is the user ID



Decode from Base64 format

Simply enter your data then push the decode button.

```
N29vck0wT24zMjEw
```

For encoded binaries (like images, documents, etc.) use the file upload form a little further down on this page.

UTF-8 — Source character set.

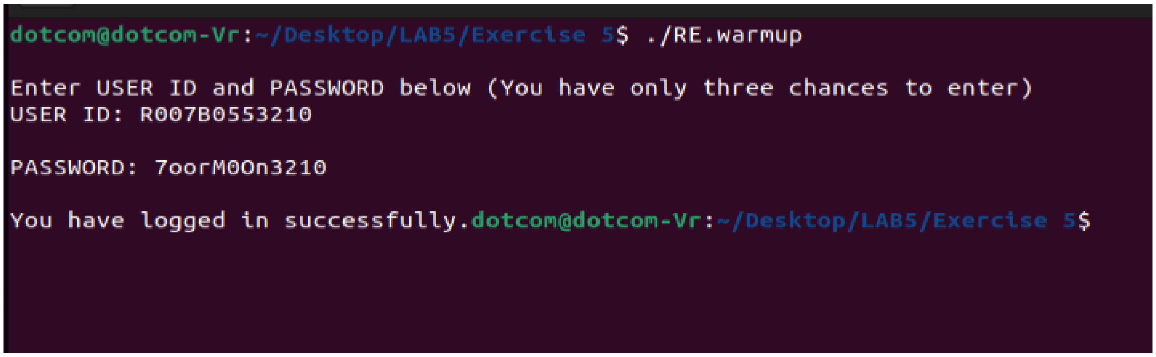
□ Decode each line separately (useful for when you have multiple entries).

Live mode OFF — Decodes in real-time as you type or paste (supports only the UTF-8 character set).

< DECODE > — Decodes your data into the area below.

```
7oorM0On3210
```

- ➔ Password is: **7oorM0On3210**
- ➔ We try to enter these :



```
dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise 5$ ./RE.warmup

Enter USER ID and PASSWORD below (You have only three chances to enter)
USER ID: R007B0553210

PASSWORD: 7oorM0On3210

You have logged in successfully.dotcom@dotcom-Vr:~/Desktop/LAB5/Exercise 5$
```

4. **Propose a better way to protect the program**

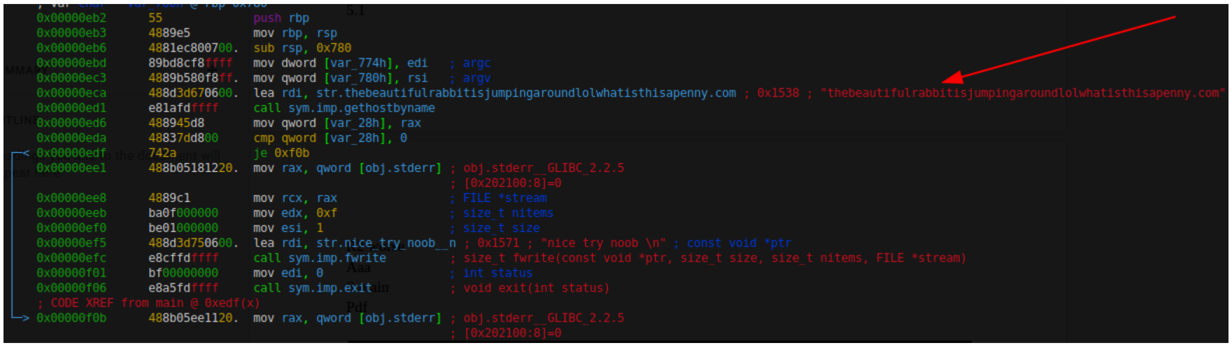
We can use the Obfuscation method to encrypt our source code so that when the attacker tries to retrieve data he will not understand it because it is encrypted.

Part II

1. **What does this malware sample do? Explain how did you find this information.**

   We installed and used Rarade2 for inspection of this malware. Follow the commands in the terminal:

   - ➔ **“r2 worse”**
   - ➔ **“aaa”** which will give options what Radare can do.
   - ➔ **“S main”**
   - ➔ **“pdf”** which will ask to print lines of the source code. Just do yes and the source code will be generated.
   - ➔ This malware program is trying to open network sockets and send DNS requests to this domain “**thebeautifulrabbitisjumpingaroundlolwhatisthisapenny.com**” and then tries to connect to the attacker’s server and send information inside **/etc/passwd**.
   - ➔ We used gdb and radare2 for tracking information leaks.



```
    0x00000eb2      55              push rbp
    0x00000eb3      4889e5          mov rbp, rsp
    0x00000eb6      4881ec80070000. sub rsp, 0x780
    0x00000ebd      89bd8cf8ffff    mov dword [var_774h], edi   ; argc
    0x00000ec3      4889b580f8ffff. mov qword [var_780h], rsi   ; argv
    0x00000eca      488d3d67060000. lea rdi, str.thebeautifulrabbitisjumpingaroundlolwhatisthisapenny.com ; 0x1538 ; "thebeautifulrabbitisjumpingaroundlolwhatisthisapenny.com"
    0x00000ed1      e81afdffff      call sym.imp.gethostbyname
    0x00000ed6      488945d8        mov qword [var_28h], rax
    0x00000eda      48837dd800      cmp qword [var_28h], 0
┌─< 0x00000edf      742a            je 0xf0b
│   0x00000ee1      488b0518122000. mov rax, qword [obj.stderr] ; obj.stderr__GLIBC_2.2.5
│                                                               ; [0x202100:8]=0
│   0x00000ee8      4889c1          mov rcx, rax                ; FILE *stream
│   0x00000eeb      ba0f000000      mov edx, 0xf                ; size_t nitems
│   0x00000ef0      be01000000      mov esi, 1                  ; size_t size
│   0x00000ef5      488d3d75060000. lea rdi, str.nice_try_noob__n ; 0x1571 ; "nice try noob \n" ; const void *ptr
│   0x00000efc      e8cffdffff      call sym.imp.fwrite         ; size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream)
│   0x00000f01      bf00000000      mov edi, 0                  ; int status
│   0x00000f06      e8a5fdffff      call sym.imp.exit           ; void exit(int status)
│   ; CODE XREF from main @ 0xedf(x)
└─> 0x00000f0b      488b05ee112000. mov rax, qword [obj.stderr] ; obj.stderr__GLIBC_2.2.5
                                                                ; [0x202100:8]=0
```

2. **In real life engagements, what steps should a malware anaylzer follow to understand a malware without being infected with it.**

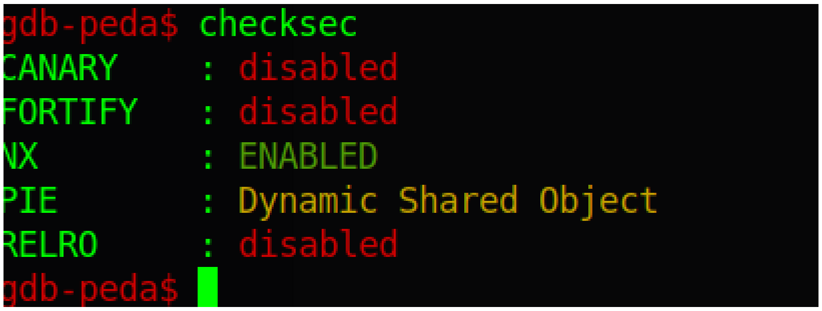
To understand a malware, a malware analyzer should follow:

The Four Steps of Malware Analysis:

1. **Fully Automated Analysis:** Static properties include strings embedded in the malware code, header details, hashes, metadata, embedded resources, etc. This type of data may be all that is needed to create IOCs, and they can be acquired very quickly because there is no need to run the program in order to see them.
2. **Static Properties Analysis:** Behavioral analysis is used to observe and interact with a malware sample running in a lab. Analysts seek to understand the sample's registry, file system, process, and network activities
3. **Interactive Behavior Analysis:** Fully automated analysis quickly and simply assesses suspicious files. The analysis can determine potential repercussions if the malware infiltrates the network and then produce an easy-to-read report that provides fast answers for security teams. Fully automated analysis is the best way to process malware at scale.
4. **Manual Code Reversing:** Using debuggers, disassemblers, compilers, and specialized tools to decode encrypted data, determine the logic behind the malware algorithm and understand any hidden capabilities that the malware has not yet exhibited.

3. What tricks and protection mechanisms does this malware implement? How did you find it?

We used GDB to find some common protection mechanisms which were used. They are:



```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : Dynamic Shared Object
RELRO     : disabled
gdb-peda$ 
```

Other than the protection mechanisms found earlier, the malware also uses the following:

- ➔ code obfuscation for making the code unreadable (using UPX)
- ➔ It sends a false DNS lookup request to confuse the analyzer.
- ➔ It does not store actual server details and passwords as a string.

We found this by analyzing the source code which we obtained by using radare2 and running them in the visual studio.

REFERENCES

1. https://github.com/gocd/docker-gocd-agent-dind/issues/18
2. https://askubuntu.com/questions/1171460/how-to-fix-docker-error-response-from-daemon-cgroups-cannot-find-cgroup-moun
3. https://www.youtube.com/watch?v=w1vMFq_iH8o
4. https://github.com/angr/angr/issues/757
5. https://pypi.org/project/angr/
6. http://spencerwuwu-blog.logdown.com/posts/1366733-a-simple-guide-of-afl-fuzzer
7. https://www.youtube.com/watch?v=O3hb6HV1ZQo
8. https://www.youtube.com/watch?v=np3sLLFQs6I
9. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=914218