**Assignment 1**

**Student name:** Mantvydas Luksas

**Student ID:** R00150390

**Question 1**

**1)**

The char buffer is allocated to have a maximum fixed size of 36 bytes. This is defined at the creation of the buffer at line 18 of the provided code. The buffer refers to the section of memory assigned to contain a value during the execution. To cause a buffer overflow we need to write more than 36 bytes into the buffer. However, this still will not be enough as the next 4 bytes will belong to the ebp address.

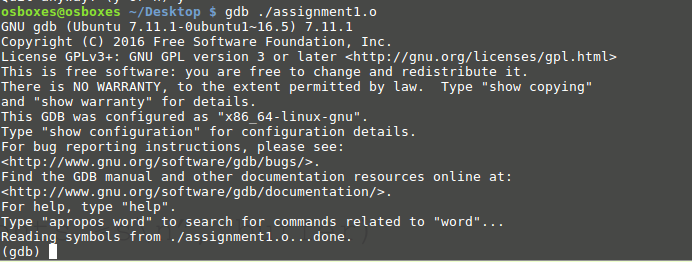
The vuln() function consisting of the creation of the buffer adds its allocation to the stack. However, before the buffer is added to the stack frame of the function, first the return address is saved when the stack frame is initially created (this is the EIP). The instruction pointer holds the address of the next instruction for the computer to execute. The EIP in this case points to the return address of main(), as when the vuln() function finishes it returns to the main() function. Of course, the EBP also known as the base pointer points to the start of a current function’s stack frame. The ebp value of the start of the new function stack frame becomes the current esp value. This is because the esp value is always pointing to the latest stack frame or to the top of the entire stack. This ebp is stored in the stack frame after the return address. Afterwards, the local variables are stored of the function and in this case the buffer needs to occupy 36 bytes, so the esp moves down 24 hex also known as 36 bytes. As each stack frame is 4 bytes, the esp moves down 9 stack frames, while the ebp is the old esp value before the allocation of the buffer size. The goal is to change the return address of the vuln() function to the return address of the partialwin(). Once at the partialwin() function we need to change its return address to point to the fullwin() function.

We know that to reach the first return address, we need to overflow the buffer by 41 bytes as the extra 4 bytes belongs to the ebp, but not the return address. The 41st byte will have an impact on the first byte of the return address stored and will cause the program to immediately crash. As each character is a byte, we need to make sure that characters after the 40th character are all characters of the partialwin() first instruction hex address.

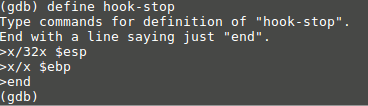
The easiest way to find out what characters need to be replaced with the return address is by running a string followed by a variety of letters, with each letter represented by a hex value in ASCII for example an A would be in hex 41. Once this is done, if there is a segmentation fault, we would be provided with the hex values that overwrote the return address. The letters represented by those hex values will be replaced with the first instruction address of partialwin() function. To replace the return address with the first instruction, we need to simply find out the address of the first instruction of partialWin() by using disassemble partialwin using gdb. To the string of 40 bytes a string representing the hex address needs to be added. This can be done using python structs. At this stage, we would not be able to write strings manually while running the program but it should be read from a file. So a python script should be written with the string that would cause the perfect buffer overflow to achieve the result. This python script will be stored in a file known as “instring”. The file will then be used while running the program, to get the string required. Once the partialwin function result displayed we need to overwrite its return address with the address of the first instruction of fullwin function.

**2)**

After the code was successfully compiled into assignment1.o, the first step was to use the GNU debugger to examine the stack. The command gdb ./assignment1.o was used.



After this I wanted to examine the contents of the 32 addresses of the stack and to do this, I simply wanted to know the first 32 addresses of the stack pointers in the stack. I wanted to examine this at each break point during debugging with gdb. In order, for this to happen a hook-stop was defined with instruction of what to display at each break point. I also wanted to display the first address of the base pointer in the stack at each break point, to find out the values contained in a current stack frame.



Afterwards, I wanted to set my break points. I decided that my first breakpoint should be just before the function vuln() was called in main() in line 25 of the code.



The second breakpoint would be at line 19 of the code before calling the gets function to get the contents of the buffer, to analyze the state of the stack with an empty buffer.

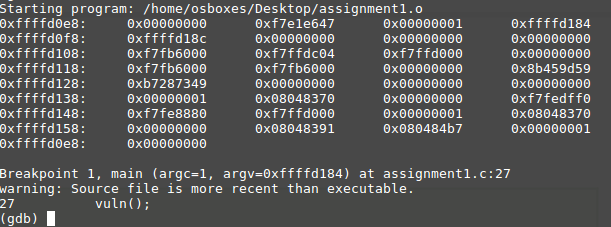


The next break point would at line 21 of the code where the value of the buffer must be already obtained. I would analyse the vuln() function stack frame to analyze the addresses at which the buffer values were stored and to view, how many are required to overwite the ebp and the return address of the stack frame.

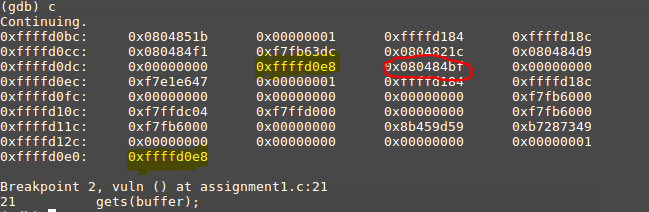
Another break is added at line 22 of the code to see the resulting buffer addresses after input and to examine the final output.



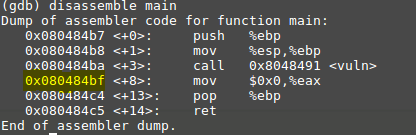
Next, I will simply run the program in the debugger by typing the run command. During this, I would end up on my first break point.



The hook-stop instructions are executed at this break point and I am given the first 32 memory addresses of stack frames within the program. The first ebp is at stack location 0xffffd0e8. We proceed with the break points by running c in the code. At the second breakpoint we can examine the stack frame of the function vuln().

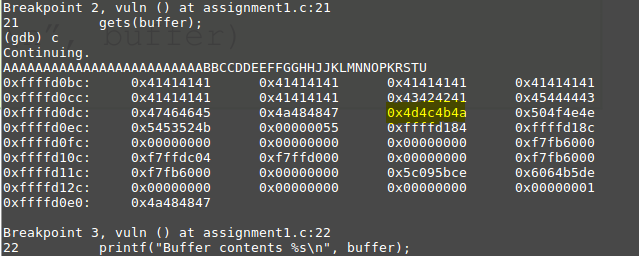


The value of the ebp is 0xffffd0e8 meaning that the current stack frame of the function is from 0xffffd0bc to 0xffffd0dc containing the values of the buffer. A total of 9 values for the buffer for 36 bytes, followed by the ebp value which is then followed by eip containing the value of the return address of main(). We can prove this by disassembling main() to find out.

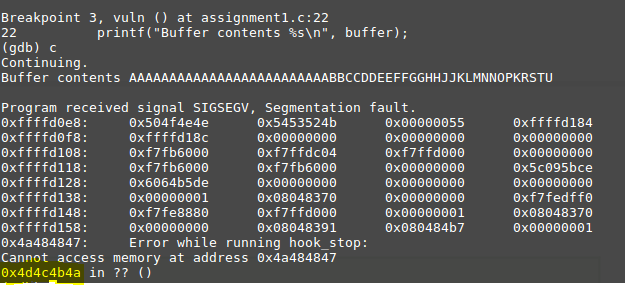


After calling the vuln() function the next instruction is exactly the same address as seen in the previous image highlighted in red. This proves that this return address in vuln() function stack frame needs to be replaced with the return address of the partialWin() function.

Having reached breakpoint 2, I was asked to provide input for the buffer and here is what I have typed in.



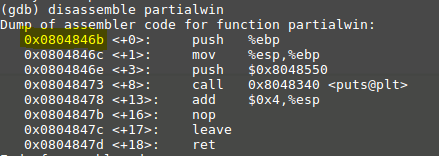
Viewing the function vuln() stack frame it is quite evident that the return address has been changed. Continuing with the running of the program, the program crashes at the next breakpoint indicating segmentation fault and the hex values of the return address that caused this to happen.



4a equates to J, 4b equates to K, 4c equates to L and 4d equates M meaning that these characters need to be replaced with the return address of partialWin().

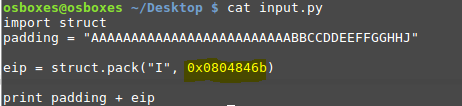


We now need to find the first instruction address of partialWin() to return to this function. This can be done by running disassemble partialWin and finding the first instruction address to be the return address.

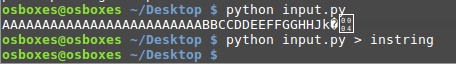


This is the first instruction address of partialwin() function and so this will be the return address at the stack frame of function vuln(). This instruction address has to be written instead of the letters “JKLM” as seen in the picture above.

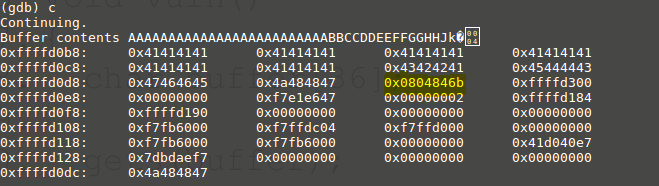
The instruction hex value can be written as a string input with the help of python structs. However, previously the string was entered manually and now I had to write a python script to output this string together with the new hex value.



I tested once again if there is a successful output and saved the script to the file instring to be used when running the program.



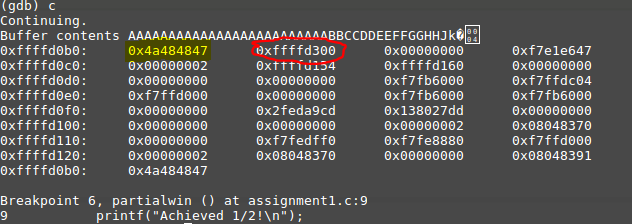
Afterwards, I could proceed with overwriting the return address to point to the first instruction in partialwin and the output was a success.



The return address was overwritten with the address of the first instruction of partialwin and the program redirected to the partialwin function.



However, this was not over yet, as I had to set another break point in the partialwin function after the print statement to examine the stack frame of the function to see how much longer my string needed to be to overwrite next return address of the partialwin function so that it would point to the return address of the fullwin function.

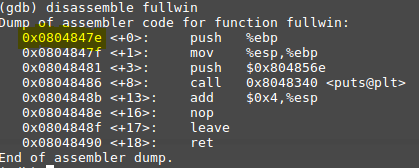


In the partial win the ebp value was already at the first address of the partialwin stack frame, following by another return address that had to be changed. This could be only done by extending our initial string by four more bytes. The last four bytes pointing to a return address of the fullwin function.

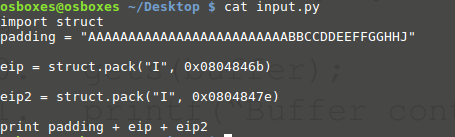
A proof of the other return address was simply stated at segmentation fault error.



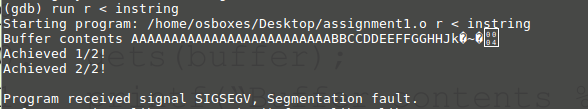
To find the return address of the fullwin() function, disassemble of fullwin had to be run again and the first instruction was taken as a return address.

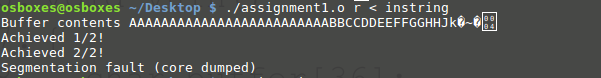


So now the final attack string looked like



Resulting in redirection from partialwin to fullwin printing both statements, while only running the program once.





Afterwards, the program crashes because it has no correct return address to return to.

**3)**

**Stack Before Attack**

Other libc functions before main is called

Main()’s stack frame

vuln()’s stack frame

Stack grows towards lower memory

**Main() function stack frame**

int argc

Stack grows high to low

EBP

Return address

char \*\*argv

**Vuln() function stack frame**

Each address Is four bytes

Stack grows high to low

0xffffd0e8

0x080484bf

Buffer[0] … Buffer[3]

Buffer[4] … Buffer[7]

Buffer[8] … Buffer[11]

Buffer[12] … Buffer[15]

Buffer[16] … Buffer[19]

Buffer[20] … Buffer[23]

Buffer[24] … Buffer[27]

Buffer[28] … Buffer[31]

Buffer[32] … Buffer[35]

EBP

Return address

**Stack After Attack**

Stack grows towards lower memory

fullwin()’s stack frame

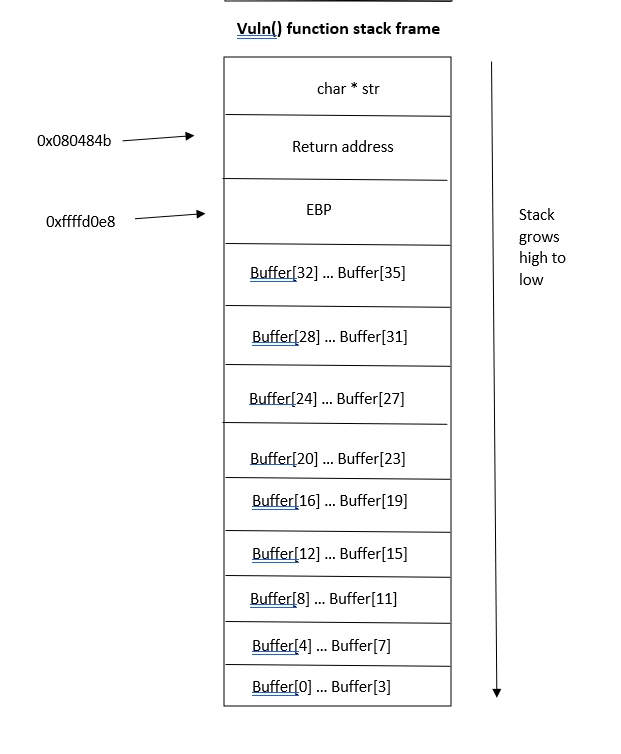
partialwin()’s stack frame

vuln()’s stack frame

Main()’s stack frame

Other libc functions before main is called

**Vuln() function stack frame**



**AAAA – 0x41414141**

Return address changed to point to address of partialwin first instruction. With eachp

**BUFFER OVERFLOW**

**~~0x080484b~~**f

**New address \x6b\x84\x04\x08 - 0x0804846b**

**AAAA – 0x41414141**

**AAAA – 0x41414141**

**AAAA – 0x41414141**

**AAAA – 0x41414141**

**ABBC – 0x43424241**

**CDDE – 0x45444443**

**EFFG – 0x47464645**

**GHHJ – 0x4a484847**

**AAAA – 0x41414141**

**partialwin() function stack frame**

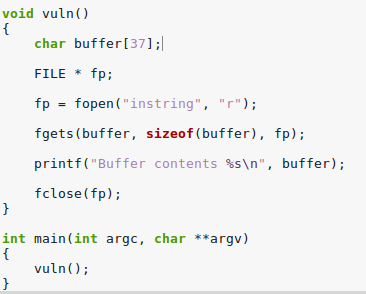
**New address \x7e\x84\x04\x08 - 0x0804847e of fullwin() function**

Return address

EBP

**BUFFER OVERFLOW CONTINUES**

**4)** Here is what has been changed in the code:

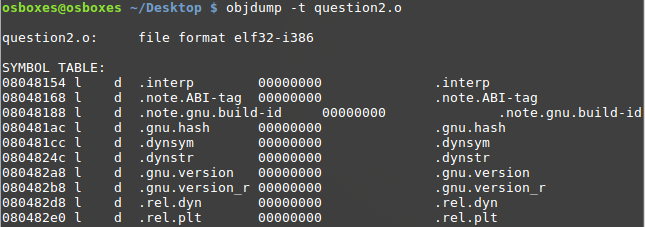


The function gets(buffer) was very dangerous to use as it continued to store characters past the end of the buffer. A safe way to store characters that does not go past the size of the buffer is by using the alternative safe function fgets. This function allows you to specify the size limit of the buffer, so that the input would never exceed the storage space of the buffer. To do this, the function takes in three parameters. The first one being the buffer itself, the second is the number of characters it can accept, and the third is the file to read the characters from. So no matter, how many characters the input file might have, only the correct amount characters will be selected to avoid a buffer overflow. Although, fgets does include the null character at the end of the string and for that reason we need to increase the size of the buffer by 1 resulting in the buffer being the size of 37 instead of 36.

**Question 2**

**1)**

**2)** After the code was successfully compiled into question2.o, the first step was to find the location of the global variable “grade” in memory so that we would know where to modify it. The “objdump” command was used to identify the objects in the executable.



Here we can find the global variables which have not been yet initialized being part of the BSS segment on the heap. BSS stands for “Block Started by Symbol”. As we look through the table, it is quite easy to spot the global variable by its “grade” and being under “.bss”.



The address of the global variable “grade” is “0804a02c”. The function “printf” will be used to change the value stored at address “0804a02c”.

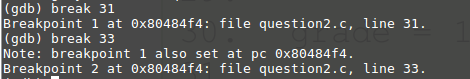
An input python script is once again needed for the program, so inputs.py was created in the following way.



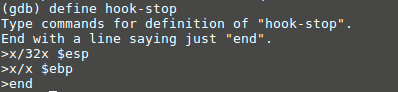
The output of the script was written to the file “inString”.



A break point is added at line 31 of the code to examine the stack frame before the input is entered. In addition, a break point is added at line 33 to examine the stack frame after the input is stored as part of the buffer.



A hook-stop is defined for every break point so that the first 32 addresses of the stack can be examined.



**Question 3**

**1)**

Address Space Layout Randomization as the name suggests randomizes the starting location for the stack and for other libraries. This is to make sure that future exploits are less possible to occur as there is a huge reliance on the buffer being always at the same memory locations so that it can be overflowed. ASLR does a great job in arranging the locations of certain data areas randomly. This includes the base of the executable and the memory locations of the stack, heap, and other libraries.