|  |  | **Cybersecurity Lab, CSE 3140** |
| --- | --- | --- |
|  |  | **Spring 2022** |
| **Memory Safety Project: The eternal war in memory** | | |
| **Section # :** | | |
| **Team #:** | | |
| **Names:** | | |

**Note: please support your answers with screenshots, steps and commands used.**

**If you write any code, backup to your local machine. This lab may cause your VM to crash. We may not be able to restore the files. So, keep all possible information in your report.**

**Change your password at your first login**

In this lab we’ll look at one of the most important computer security issues that occurs in practice, memory overreads and overwrites. This lab requires some basic understanding of how computer memory is laid out both architecturally and from the operating system perspective.  However, the basic principle is very simple, in many programming languages (particularly C), all data types are raw blocks of memory.  An integer is just 32 bits in memory that are interpreted as an integer for the purposes of the program.  This becomes particularly problematic for data types that are variable length.  As an example, if I want to create a size 10 buffer, I ask the operating system for 10 bytes of blank memory.  For example, using:

**char \* *buf*= malloc(10);**

The value returned in ***buf***points to ten continuous bytes in memory that the operating system has designated for this purpose.  However, if I write **buf[11]** the programming language doesn’t know the size of ***buf***, it is just an array in memory.  Essentially, C views memory as one large object.  This may not seem troubling yet; we are getting there.  Since items don’t have attached sizes it is very easy for these “boundaries” between objects to get crossed.

You’ll be doing all of the work on a group VM, you will log into [cse@172.16.50.X](mailto:cse@172.16.50.XY) ( X= 20\*section number  + group number) with username: ***cse*** and password: ***cse3140***.

**Question 1 (10 points):** Consider the following code:

int awesome(){  
 int a[10];  
 int b[20];  
 for(int i=0;i<10;i++){ a[i] = 0;}  
 for(int i=19;i>=0;i--){ b[i] = i; }  
 int i=19;  
 while(b[i]>b[i-1]){   
 a[i] = b[i];   
 i--;  
 }   
 return i;  
}

What do you think should be the behavior of this code? Should it be executed? Should a compiler fail?   
  
project-group-assignment

Suppose for a moment that a and b are laid out sequentially in memory. That is,

\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|  
a[0]a[1] a[2] a[3] a[4] a[5] a[6]a[7] a[8]a[9] b[0] ….. b[19]

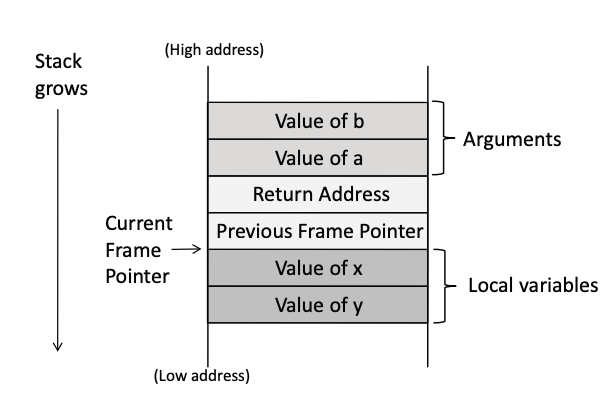
are 30 consecutive blocks. Then what is the behavior of the above code? What is the value of i output by the program?

Throughout this lab use the **-g and -m32** switches for gcc when building code. This will make your life easier. The -g flag specifies to include debugging information. The -m32 flag specifies to build code for a 32-bit architecture (rather than a 64-bit executable which is the default for most systems). If you build 64-bit executables, most of the descriptions of the lab will not make sense. Rebuild your code using those switches, does the answer change?

From how the program behaves, is the above picture correct about memory being laid out? If there are any differences, pose one reason for this possible difference:

**Question 2 (10 points):** The other treacherous ingredient that comes into play is the way the stack works in computer architecture. When a function is being executed and wants to call another function there is a memory data structure called the stack in which three things are placed:

1. The current function context including registers and what function was being executing (so the program knows where to go back to when the called function finishes)
2. Arguments for the function to be called.
3. All the local variables initialized in the called function.



What is the primary purpose of the stack-based architecture? What types of programs does it enable?

This is shown visually above. [Another resource is here](https://www.cs.princeton.edu/courses/archive/spr11/cos217/lectures/15AssemblyFunctions.pdf). Importantly, the addresses decrease as the stack grows. What are frame pointers and return addresses? What are they used for?

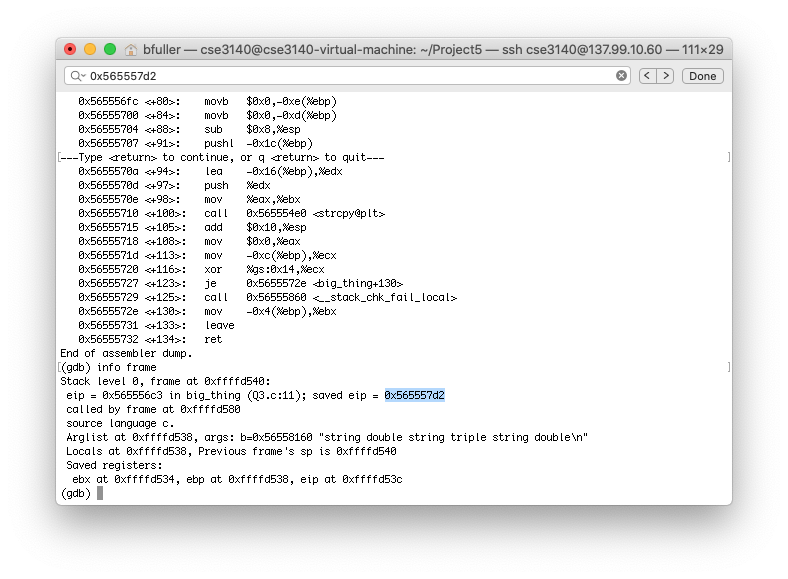
On your virtual machine is a snippet of code called Q2.c, compile and execute this code. **Throughout this lab use the -g and -m32 switches for gcc when building code. This will make your life easier. The -g flag specifies to include debugging information. The -m32 flag specifies to build code for a 32-bit architecture (rather than a 64-bit executable which is the default for most systems). If you build 64-bit executables, most of the description of the lab won’t make sense.**

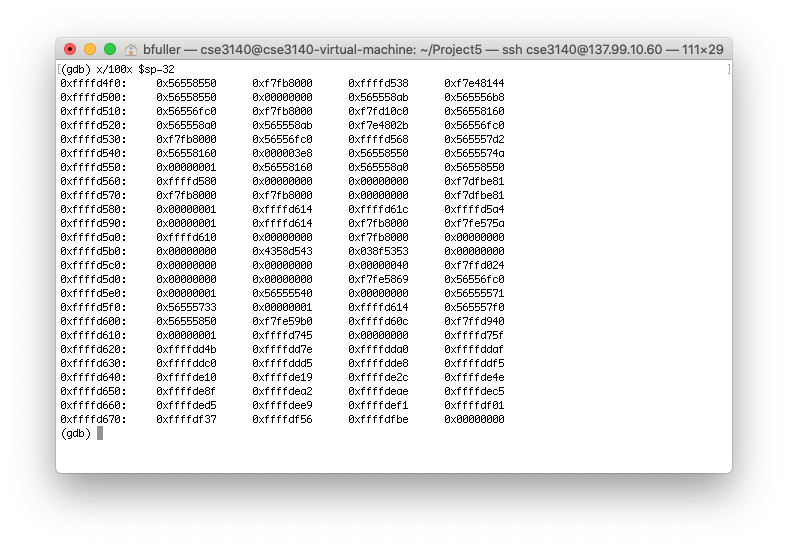
Explain why the code is crashing. What is the minimum size of ***size\_to\_use*** that causes the program to fail? (Test this programmatically.)

Remove the commenting for line 14, which prints the address of ***size\_to\_use***, and change the value of size\_to\_use to be set to (2). Run the program and observe the address of **size\_to\_use** variable. What is the address location of **size\_to\_use**? Rerun the program multiple times, what do you notice? This phenomena is because there is an implemented OS countermeasure named [Address space layout randomization](https://en.wikipedia.org/wiki/Address_space_layout_randomization) (ASLR). Why do you think it may be helpful?

What command should enable or disable this countermeasure in the OS? ***Ask you TA to disable this countermeasure for you. How can you check that the countermeasure is disabled on your VM?***

**Question 3 (20 points):** Putting the issues in the last two questions together makes it clear that some languages make it easy to have memory corruptions which cause issues. However, the problem is much bigger. Since the return address is written on the stack, it is possible for that to be overwritten. This corrupts the control flow of the program and moves these issues from annoying to major problems. For this question, you’ll have to use a debugger to get things moving. We’ll be using [gdb](http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf). A nice video on how to analyze the program using gdb can be found [here](https://www.youtube.com/watch?v=V9lMxx3iFWU). You are going to run **Q3** and set a breakpoint at the function **big\_thing**. (Remember to compile with **-m32 -g**). Run the program to this point. You should now be able to see the stack using ***backtrace*** command. You should see two entries on the stack trace, one for the main program and the second called function **big\_thing**e memory address of the previous stack frame. Find these two ways: **info frame** and **x/100x $sp-32**. The second command shows you 256 bytes starting 32 bytes above the current stack pointer. What is the current address for the previous frame’s **eip**? (This is called the saved eip below.) At what address on the stack is this written?





What ten bytes are used for storing a? You can print &a in gdb. **A small warning: in gdb you may see bytes reordered from the position you think they should be in. Ten consecutive bytes of zeros may not appear that way as the stack grows down.**

How many bytes are between where a is stored and the previous instruction pointer? Do your best to identify the purposes of each byte.

Step until the strcpy executes. How has the stack changed? What has been overwritten into the previous eip?

Step in gdb until you see an error ***stack smashing detected***. This is a protection called [canaries](https://en.wikipedia.org/wiki/Stack_buffer_overflow#Stack_canaries) put in place by gcc. Explain what canaries are below?

Based on your explanation of canaries, how would you go about creating an exploit that works with canaries in place?

To understand the core behavior, we’re going to turn off this protection using the flag -fno-stack-protector. **Note: do not use this flag in development.** Re-execute the program. How many bytes are now between an array and the stored eip?

Step until after the strcpy command. What value has overwritten the return address for the previous instruction pointer? You can access this through either method above. What characters of the text file does this correspond to? Notice the order of characters.

Step through the return of the function, where does the program flow try to go? What happened? Once the program jumps, hit continues.

You’re now going to write your first actual exploit. Modify test.txt so that you jump to location 0x**deadbeef**. In what position did you change your character?

**Question 4 (25 points):** You’re still working with Q3 code. There is a function there that we haven’t called yet called secret\_function. Launch gdb, set a breakpoint on **big\_thing** and find the address for the secret\_function. Now rewrite your test.txt to jump to this location. You may have to reorder bytes with some trial and error (see [endianness](https://en.wikipedia.org/wiki/Endianness)). What was the proper byte order to get the string to print?

Close the program and run it again, did you get the secret string to print with the same input? Gdb and the underlying operating system may use [Address space layout randomization](https://en.wikipedia.org/wiki/Address_space_layout_randomization). Explain this technique in a paragraph below. You can disable the address space randomization when debugging Linux applications using a **set disable-randomization on** command**. Again, don’t use this in practice.**  With this disabled, is the address of **secret\_function** the same every time? Can you write a script that works predictably? Why or why not?

**Note that the addresses may change inside and outside of gdb so it's fine if you only print inside of gdb. Also segmentation faults may cause print statements to not appear so step through your code.**

Change your input file to launch a shell. Look carefully at the resources available to you in Q3.c. You should not need to modify the source code. Describe what you did to make this change:

We are almost implementing the exploit. In reality, you do not have a secret\_function that you are trying to find and execute. But, you now understand how an attacker can hijack the execution of the program and jump to any location. If the program is privileged, the attacker will gain escalated privilege. We now need to get our code (i.e. malicious code) into the memory of the running program. We can simply place our code in the **test.txt** which is inputted to the program. So, when the file is copied to the stack our code will be placed in the stack. The next step is to force the program to jump to the exact location of our malicious code. You can now do that as you learned how to override the return address.

In theory, this is how buffer over attack works. In practical terms, there are many complications behind that. The way programs are executed and the implemented countermeasures make this scenario a little bit harder. One problem is that the stacks are non-executable by default. Which prevents the injected code from getting executed. This countermeasure can be defeated by the Return-to-libc attack which is part of a concept called Return-Oriented Programming. For now let’s disable this protection by compiling the source code with the option ***–z execstack*** added. In addition we still have the two countermeasures disabled from before. Keep the ***–fno-stack-protection –m32 and –g options while compilation.***

To be able to jump to our malicious code, we need to know the memory location for the placed code in the stack. In this experiment we have the source code for the program so we can compile, then debug using gdb. You can get the address of the frame pointer and the buffer address to correctly place the jump to address, in your return address.

In your buffer folder. There is shellscript.py which can help creating a badfile.txt, containing a script to exploit a shell. The given ***shellcode*** in the file is simply a copy of a malicious code. You can check online how to write this code. You can also try different codes if you wish. We also make use of an operation named NoP (0x90). Why do you think we need such an operation?

If you are not able to get the code to work just describe with screenshots what you’d need to do for this process to be successful.

Question 5 (10 points): One last defense we haven’t considered in this lab is a protection known as W^X which specifies that each page of memory should be either writable or executable but not both. This defense can be defeated using [return oriented programming](https://en.wikipedia.org/wiki/Return-oriented_programming). Describe the return-oriented programming below.

Present one possible defense against this attack, what are some disadvantages of the defense you presented?

Question 6 (15 points): You’ll be looking at an unknown binary without binary. This question reads from ***q6answer.txt***. You need to write code to get these programs to jump to **you\_win** which takes no arguments. This function will print out a flag for you to record below. These programs follow the same basic structure as **Q3.c. However**, there are several extra checks on the value of **b**, so you can’t just directly use an arbitrary string for the attack. Your goal is to figure out what checks you need to pass and form a string that effectively passes all these checks. Some helpful commands, running info files will tell you about the different points in memory. Entry point is the address of main. We recommend adding a breakpoint at that location.

Once in main you can type disassemble to see the code that is being run. Gdb will annotate calls with functions that it knows. You should be able to identify calls to fgets, strlen, etc. There’s one call that is of particular interest. This is the function check\_string. Its assembly is below. You have two tasks at this point. The first is to understand what requirements this is placing on the string. The second is to build a file that causes Q6 to jump to the you\_win function. Below is an example disassembly of the check\_string function. We provide some hints below at the assembly. You may wish to use consult information [x86 architecture](https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture) and specific [instructions](https://en.wikipedia.org/wiki/X86_instruction_listings#Original_8086/8088_instructions). I would encourage you to add your interpretation of the code inline (or to write a separate function with your guess and disassemble that function). Some instructions may be easier to interpret than others.

0x565556f2 <+0>: push %ebp

0x565556f3 <+1>: mov %esp,%ebp

0x565556f5 <+3>: call 0x56555988 <\_\_x86.get\_pc\_thunk.ax>

0x565556fa <+8>: add $0x18c2,%eax

0x565556ff <+13>: pushl 0xc(%ebp)

0x56555702 <+16>: pushl 0x8(%ebp)

0x56555705 <+19>: call 0x5655568d <is\_palindrome>

0x5655570a <+24>: add $0x8,%esp

0x5655570d <+27>: test %eax,%eax

0x5655570f <+29>: jne 0x5655571b <check\_string+41>

0x56555711 <+31>: mov $0x1,%eax

0x56555716 <+36>: jmp 0x56555801 <check\_string+271>

0x5655571b <+41>: mov 0x8(%ebp),%eax

0x5655571e <+44>: movzbl (%eax),%eax

0x56555721 <+47>: movsbl %al,%edx

0x56555724 <+50>: mov 0x8(%ebp),%eax

0x56555727 <+53>: add $0x1,%eax

0x5655572a <+56>: movzbl (%eax),%eax

0x5655572d <+59>: movsbl %al,%ecx

0x56555730 <+62>: mov 0x8(%ebp),%eax

0x56555733 <+65>: add $0x2,%eax

0x56555736 <+68>: movzbl (%eax),%eax

0x56555739 <+71>: movsbl %al,%eax

0x5655573c <+74>: imul %ecx,%eax

0x5655573f <+77>: cmp %eax,%edx

0x56555741 <+79>: je 0x5655574d <check\_string+91>

0x56555743 <+81>: mov $0x1,%eax

0x56555748 <+86>: jmp 0x56555801 <check\_string+271>

0x5655574d <+91>: mov 0x8(%ebp),%eax

0x56555750 <+94>: add $0x5,%eax

0x56555753 <+97>: movzbl (%eax),%eax

0x56555756 <+100>:cmp $0x68,%al

0x56555758 <+102>: je 0x56555764 <check\_string+114>

0x5655575a <+104>: mov $0x1,%eax

0x5655575f <+109>: jmp 0x56555801 <check\_string+271>

0x56555764 <+114>: mov 0x8(%ebp),%eax

0x56555767 <+117>: add $0x6,%eax

0x5655576a <+120>: movzbl (%eax),%eax

0x5655576d <+123>: cmp $0x61,%al

0x5655576f <+125>: je 0x5655577b <check\_string+137>

0x56555771 <+127>: mov $0x1,%eax

0x56555776 <+132>: jmp 0x56555801 <check\_string+271>

0x5655577b <+137>: mov 0x8(%ebp),%eax

0x5655577e <+140>: add $0x7,%eax

0x56555781 <+143>: movzbl (%eax),%eax

0x56555784 <+146>: cmp $0x63,%al

0x56555786 <+148>: je 0x5655578f <check\_string+157>

0x56555788 <+150>: mov $0x1,%eax

0x5655578d <+155>: jmp 0x56555801 <check\_string+271>

0x5655578f <+157>: mov 0x8(%ebp),%eax

0x56555792 <+160>: add $0x8,%eax

0x56555795 <+163>: movzbl (%eax),%eax

0x56555798 <+166>: cmp $0x6b,%al

0x5655579a <+168>: je 0x565557a3 <check\_string+177>

0x5655579c <+170>: mov $0x1,%eax

0x565557a1 <+175>: jmp 0x56555801 <check\_string+271>

0x565557a3 <+177>: mov 0x8(%ebp),%eax

0x565557a6 <+180>: add $0x9,%eax

0x565557a9 <+183>: movzbl (%eax),%eax

0x565557ac <+186>: cmp $0x65,%al

0x565557ae <+188>: je 0x565557b7 <check\_string+197>

0x565557b0 <+190>: mov $0x1,%eax

0x565557b5 <+195>: jmp 0x56555801 <check\_string+271>

0x565557b7 <+197>: mov 0x8(%ebp),%eax

0x565557ba <+200>: add $0xa,%eax

0x565557bd <+203>: movzbl (%eax),%eax

0x565557c0 <+206>: cmp $0x72,%al

0x565557c2 <+208>: je 0x565557cb <check\_string+217>

0x565557c4 <+210>: mov $0x1,%eax

0x565557c9 <+215>: jmp 0x56555801 <check\_string+271>

0x565557cb <+217>: mov 0x8(%ebp),%eax

0x565557ce <+220>: add $0xb,%eax

0x565557d1 <+223>: movzbl (%eax),%eax

0x565557d4 <+226>: movsbl %al,%eax

0x565557d7 <+229>: mov 0x8(%ebp),%edx

0x565557da <+232>: add $0xc,%edx

0x565557dd <+235>: movzbl (%edx),%edx

0x565557e0 <+238>: movsbl %dl,%ecx

0x565557e3 <+241>: mov 0x8(%ebp),%edx

0x565557e6 <+244>: add $0xd,%edx

0x565557e9 <+247>: movzbl (%edx),%edx

0x565557ec <+250>: movsbl %dl,%edx

0x565557ef <+253>: add %ecx,%edx

0x565557f1 <+255>: cmp %edx,%eax

0x565557f3 <+257>: je 0x565557fc <check\_string+266>

0x565557f5 <+259>: mov $0x1,%eax

0x565557fa <+264>: jmp 0x56555801 <check\_string+271>

0x565557fc <+266>: mov $0x0,%eax

0x56555801 <+271>: leave

0x56555802 <+272>: ret

What is this code doing? What are the requirements for the string you enter? Present your string below as well as the challenge flag when you correctly pass a string below.

Hints:

1. There are many jumps to 0x565557fc, this is the end of the function. This is what happens when you return in the middle of a function.
2. There are six general purpose registers that are used for computation: EAX, EBX, ECX, EDX, ESI, and EDI. In addition, EBP is used to indicate the base of the stack and ESP is used to indicate the current stack pointer. This changes with pop, push, call, and ret.
3. The returned value is placed into %eax. So having an instruction of mov $0x1, %eax right before a return means the function is returning 1.
4. Mov indicates a move. The different move instructions are for different types of data. For example, movzbl means move a byte from the first location to the second location which is a long (32 bits) and zero extend.
5. The function \_\_x86.get\_pc\_thunk.ax is used for generating position independent code and is not crucial for your string.
6. Parenthesis around a value are a memory lookup. (%eax) represents the memory address in the location currently stored in %eax.
7. %al is the bottom 8 bits in %eax
8. The pointer for b is stored at 8(%ebp).

Question 7 (10 points): You may think that everything described in this lab is just an artifact of the C programming language, and no sane programmer uses C. How would you change the design of a programming language to stop such corruption? Keep in mind all of the things coming together to create these problems: memory corruption, stack architecture, and processing of untrusted data.

Look at the [top 10 programming languages](https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019), note that C and C++ are in the top 10. Why do people continue to use these languages, list at least 3 reasons.