Lab 5

Objective:

- The purpose of this lab is to allow us to experiment with Stack Randomization on our ***target machine***.

Lab 5 Submission – Participation Activity 7

- In this Lab 5, you will need to submit your Participation Activity 7:
    - Pick up a copy of the last page of this Lab 5, i.e., Participation Activity 7, from the TA during your lab session.
    - Once you have completed this lab and your Participation Activity 7, write your name, student number and circle your section (D100 or D200) and hand it in to the TAs during your lab session on either Monday March 13 or Tuesday March 14.

Participation Activity 7 Deadline: If you have not already done so, you must hand in your PA7 in paper form by the end of the last lab session (i.e., 12:20pm) on Tuesday March 14.

No late or electronic submission of PA7 will be accepted.

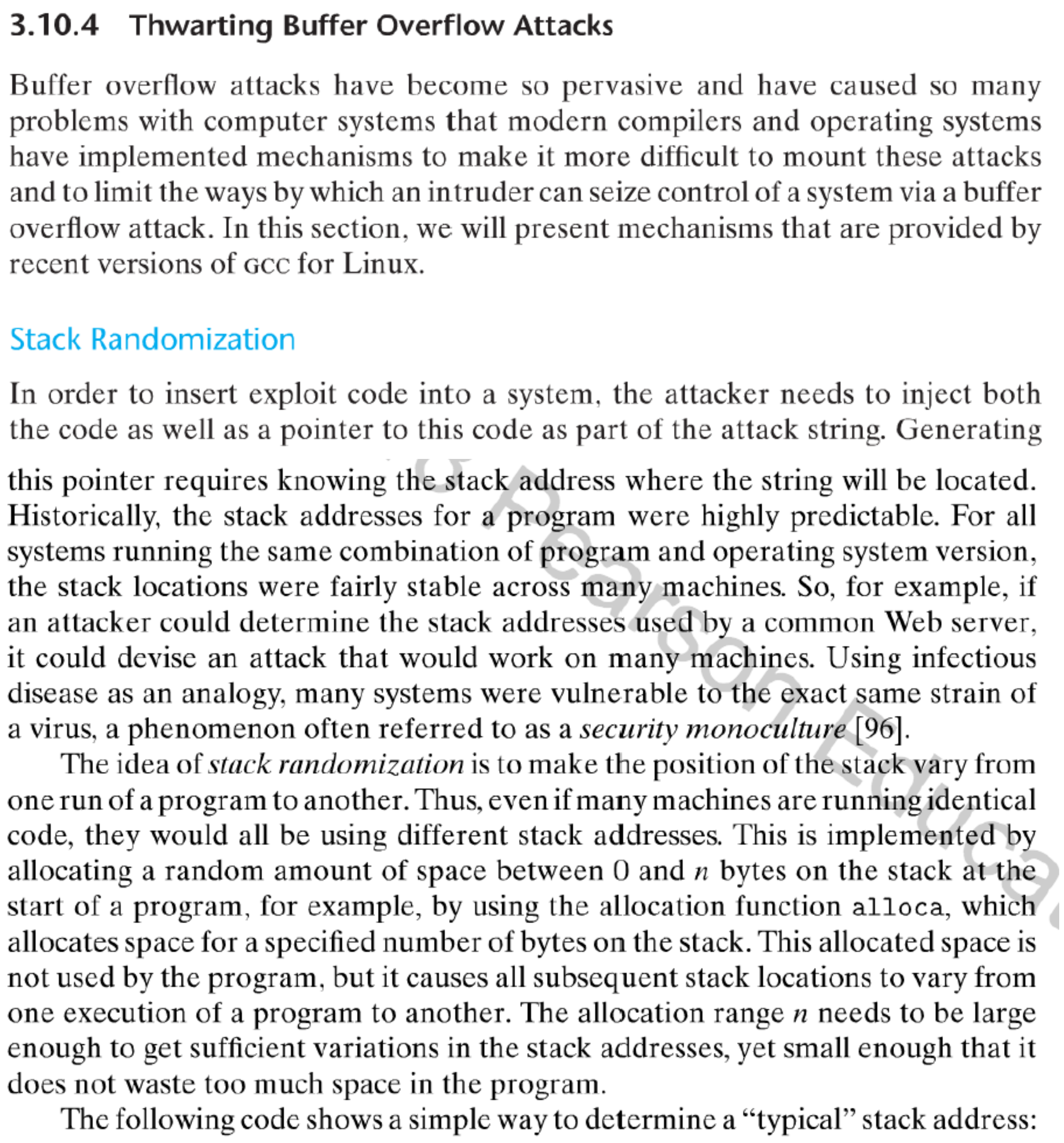
We saw in our Lecture 22 that one of the ways to counter buffer overflow attacks was to employ system-level protections such as address-space layout randomization (ASLR).

> From Wikipedia: **Address space layout randomization** (**ASLR**) is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities. In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries. Thank you, Wikipedia!

In other words, when a program starts executing, the system first allocates a random amount of space on the stack which is different every time the executable executes.

The **effect** of such technique is to shift the stack addresses for the entire program, i.e., shift the memory address of all the stack frames created by each function of the program as they are called and executed. Since this random offset (or displacement) is different every time the program executes, it makes it difficult for hackers to guess the start memory address of each stack frame where they could inject ***malicious executable code*** and therefore guess the return memory address to their malicious executable code once it has been injected into the stack frame of the executing function

Here is what our textbook has to say on the topic of Stack Randomization (Section 3.10.4):



3.10.4 Thwarting Buffer Overflow Attacks

Buffer overflow attacks have become so pervasive and have caused so many problems with computer systems that modern compilers and operating systems have implemented mechanisms to make it more difficult to mount these attacks and to limit the ways by which an intruder can seize control of a system via a buffer overflow attack. In this section, we will present mechanisms that are provided by recent versions of GCC for Linux.

Stack Randomization

In order to insert exploit code into a system, the attacker needs to inject both the code as well as a pointer to this code as part of the attack string. Generating this pointer requires knowing the stack address where the string will be located. Historically, the stack addresses for a program were highly predictable. For all systems running the same combination of program and operating system version, the stack locations were fairly stable across many machines. So, for example, if an attacker could determine the stack addresses used by a common Web server, it could devise an attack that would work on many machines. Using infectious disease as an analogy, many systems were vulnerable to the exact same strain of a virus, a phenomenon often referred to as a *security monoculture* [96].

The idea of *stack randomization* is to make the position of the stack vary from one run of a program to another. Thus, even if many machines are running identical code, they would all be using different stack addresses. This is implemented by allocating a random amount of space between 0 and n bytes on the stack at the start of a program, for example, by using the allocation function `alloca`, which allocates space for a specified number of bytes on the stack. This allocated space is not used by the program, but it causes all subsequent stack locations to vary from one execution of a program to another. The allocation range n needs to be large enough to get sufficient variations in the stack addresses, yet small enough that it does not waste too much space in the program.

The following code shows a simple way to determine a "typical" stack address:

```
int main() {
    long local;
    printf("local at %p\n", &local);
    return 0;
}
```

This code simply prints the address of a local variable in the main function. Running the code 10,000 times on a Linux machine in 32-bit mode, the addresses ranged from 0xff7fc59c to 0xffffd09c, a range of around 2^{23}. Running in 64-bit mode on the newer machine, the addresses ranged from 0x7fff0001b698 to 0x7ffffffaa4a8, a range of nearly 2^{32}.

Stack randomization has become standard practice in Linux systems. It is one of a larger class of techniques known as *address-space layout randomization*, or ASLR [99]. With ASLR, different parts of the program, including program code, library code, stack, global variables, and heap data, are loaded into different regions of memory each time a program is run. That means that a program running on one machine will have very different address mappings than the same program running on other machines. This can thwart some forms of attack.

In this lab you are asked to confirm or disprove the fact that our ***target machine*** does implement Stack Randomization by running your own experiment as follows:

- Download from our course web site (see Lecture 22) the password program (**`password.c`**) we saw as part of our Lecture 22 demo.
- Add a statement that prints the memory address of the first byte of the array **`password`**. You may want to refer to the section of our textbook included in this lab above for some inspiration as to the C statement(s) you are asked to add to **`password.c`**.
- Build the executable using the **`makefile`** provided on our course web site, with or without the flag **`-fno-stack-protector`**.
- Execute your program 10 times and record the memory addresses at which this local variable **`password`** (array) is stored on the stack. Record these memory address in the table found on the last page of this lab.
- Calculate the ***range of variance*** in the memory addresses you obtained (if any) over the 10 executions of your program. Again, please, refer to the section of our textbook included in this lab above for an example of what a ***range of variance*** could look like. On the last page of this lab, express this ***range of variance*** in bytes and using its decimal value.

- Finally, state your conclusion which must be supported by your experiential results. Do so by circling the word ***confirm*** or the word ***disprove*** in the conclusion found on the last page of this lab.

Last (Family) name:

First name:

Student number:

Circle your section: D100 D200

Lab 5 - Participation Activity 7

Execution #	**Memory address of local variable `password`**
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

The ***range of variance*** is ____ bytes

Show the work you needed to do in order to compute your answer above:

Conclusion:

I confirm or disprove the fact that our ***target machine*** does implement Stack Randomization.