Stack Smashing Summary

Reading through these articles was a nice refresher on some of the fundamentals of stack smashing. Prior to this class, I was a teaching assistant for 224, which is the class where we teach about stack smashing/buffer overflow (as well as many other things.) As a result of my experience as a TA, I am fairly familiar with the concepts underpinning the buffer overflow attack. I've learned that the best way to execute these types of attacks is to diagram the layout of memory through the function's lifecycle, then determine what kinds of inputs you can give to cause unexpected behavior.

Computation has many abstract concepts which we can use to help decompose our code and make it more clear. One of these concepts is known as a 'function'. A function is a group of code that takes in some inputs and produces some outputs. A function does not know (or care) about who called it, and is only concerned with converting the inputs into outputs. The function's apathy about its caller means that the function itself stores no data about who called it. This forces the computer to store information about who called whom on the stack. The stack is just an area of memory which can store values and grows 'down' as it expands (though 'down' is meaningless in the void of a computer). When we use certain low level functions such as gets of scanf, we can unwittingly allow a user to overwrite the stored information about the function caller. The overwriting of this information can cause the program to jump to a nearly arbitrary location in memory. This attack is known as 'stack smashing'

Modern architectures have safeguards against this sort of thing, including 'stack canaries', marking the stack as non-executable, and randomizing the location of things in memory. By using these three constructs, the typical code injection/stack smash attack becomes much harder to execute successfully. The stack canary prevents the stack from being manipulated through a construct similar to a freshness seal. Imagine it like this, when a function is called, the computer leaves behind a line of sand in a particular pattern which is difficult to guess. Then when it is walking backwards (exiting the function call), it checks the pattern in the sand to see if it has been disturbed. If the sand is not how it should be, stack smashing may have occurred and the program will exit. The other two counter measures also serve to defeat the injection attack, but in different ways.

I have a few questions that I hope can be answered in class about this type of exploit.

- Given modern protections, are these kinds of attacks viable?
- Why can't people switch away from C so they get memory safety? Is speed worth your sanity?