Problem 1 In class, we discussed Ken Thompson’s paper, “Reflections on trusting trust,” in which he described a technique for installing an undetectable login backdoor by adding a second backdoor to the compiler. The backdoored compiler inserts the appropriate backdoors when compiling the login program and the compiler itself. Once the binary of the compiler, used for bootstrapping future systems, implements the backdoor, any trace of tampering can be removed from the source. In this problem, we will explore a technique for detecting such an attack.

Assume we have two C-language compilers, for example GCC and MSVC. We suspect that nefarious hackers have inserted the Thompson backdoor into the GCC binary on our Linux system, but we believe that these hackers are believers in the free-software movement, and so wouldn’t cooperate with Microsoft’s compiler division. It’s impractical to build an entire Linux system with MSVC, since Linux programs are written to expect GCC’s extensions to the C language. But we should be able to get MSVC to build GCC; though this would most likely require some modification to GCC.

Describe how we can reliably detect the presence of a GCC backdoor by building GCC with MSVC.

Problem 2 On the x86 and most other processor architectures, the stack grows downward: push instructions, and other instructions that add to the stack, like call, decrease the stack pointer. This is just a convention: it is possible to have a computer in which the stack grows upward: a push instruction has the effect of increasing the value of the stack pointer. Only the compiler needs to change. Indeed, the stack grows upward on a few architectures, notably HP’s PA-RISC.

(a) Show how the function-call ABI would have to change if the x86 stack grew upward. That is, give the assembly the compiler will issue (1) in the caller, when it makes the function call; (2) in the callee, when it begins; (3) in the callee, when it finishes; and (4) in the caller, when control returns to it after the callee finishes. (Items (2) and (3) are often called the function prologue and epilogue.)

(b) Are stack-smashing buffer-overflow attacks of the sort we discussed in class still feasible when the stack grows upwards? Explain, including memory layout diagrams and sample code where useful.

Problem 3 In class we discussed “canaries” or “stack cookies” as a way to detect a buffer overflow on the stack and stop the program’s execution before the attacker can take
it over. In this problem, we will consider different choices in the implementation of stack cookies and how they affect the security benefits of the defense. For each part, you should explain whether the design choice makes a difference to the security of the resulting system.

More precisely, if there is no security difference, briefly explain why. If there is a security difference, show it: Give a program that is vulnerable to attack under one design choice but not under the other, and explain how the attack would proceed in the one case and why it would not be possible in the other.

(a) Does it matter whether the stack cookie is placed immediately above the saved frame pointer (and below the saved instruction pointer) or immediately below (and above the local variables)?

(b) Does it matter whether, in addition to placing the stack cookie in the optimal location according to part (a), the local variables in each function are reordered so any array local variables are placed below any non-array local variables? (In this case, proceeding from lower addresses to higher, a stack frame would consist of: non-array local variables; array local variables; the stack cookie and saved frame pointer, in whatever order; and the saved instruction pointer.)

(c) Assuming the stack cookie is a random four-byte value chosen independently for each process (as opposed to a fixed “terminator” value), does it matter whether this value is stored in memory (say, in the program’s data segment) or in a dedicated register?