Chris Cohen
CS354 Spring 2020
Lab 5 - Due 4/21/2020

BONUS

1) In Problem 4, lab4, an attacker modified the return address of ctxsw() to jump to its malware code which then behaved overtly or covertly depending on the attacker's objective. In the bonus problem, consider mounting a defense in the context of XINU running on galileo backends subject to the attacker of lab4. The solution cannot use a canary to detect corruption since return address overwrite was performed surgically without collateral corruption of surrounding memory. Instead, think about a ROP based defense where kernel code is modified to check if the return address of ctxsw() (and similarly for resched() and sleepms()) is valid, i.e., safe to jump to, and does so only if it is. Perhaps a modified return address may even be corrected to its original value so that jumping is feasible despite an attack. Describe a detailed solution in Lab5Answers.pdf in lab5/. There is no need to implement your solution.
   a) **The first thing to try is what we did in this lab to verify callback function integrity. Simply put, if the return address is not within the text section, something has gone wrong, so terminate the program. This comes with the disadvantage of the program not completing execution, though.**

      If the return address is overwritten, there is no way to get it back and recover from the attack unless there are duplicates in the stack. One way to recover is to simply push the address twice, and every time you execute a ret instruction, compare the two return addresses. If these addresses are right next to each other, they may both be changed by the attacker, which would defeat the purpose of the duplicates. I propose pushing a duplicate value onto the stack before, for example, EBP/EFLAGS/EIP/etc at the end of a function call. The important part is that the duplicate value is in a known location, but also tougher to find than the typical place for a return address.

      Similar to how XINU explicitly defines and writes system calls (like memcpy(), printf(), strchr(), etc) in ‘lib/’, we could explicitly define how the ret instruction works. Instead of a typical ret instruction, the kernel would execute the code to

verify the return address, correct if needed, then jump to the original return address of ret.