1. Write a C program which take into account two integers A, B and output the greatest common divisor of both A and B. Example: A = 10, B = 15 then output 5; A = 7, B=19 then output 1.

**The program prompts the user to input the dividend (integer A) and divisor (integer B) and prints the result (the greatest common divisor of both). If the divisor entered is zero, the program re-prompts the user to enter a new divisor. The program displays the computed greatest common divisor as a positive integer even if one or both of the integers entered are negative. File name: Snyder\_HW\_01\_01.c**

1. Write a C program which asks the user for an integer N. Output the prime number P that is closet to N and is greater than N. Example: 10 then output P = 11; N = 23 then output P=29.

**The program prompts the user to input an integer and prints the result (the closest prime number greater than the input integer). File name: Snyder\_HW\_01\_02.c**

1. Write a C program that allows the user to enter two integers and output their sum and product. Observe the behavior of your program when the two integers were big.

**The program prompts the user to input two integers and prints the result (the sum and product of the input integers). When both integers entered were large, the program gave the sum as -2 and the product as -1. File name: Snyder\_HW\_01\_03.c**

1. Write a “*Hello World*” C program with some extra initialized and uninitialized variables. This program will also be used in questions 7, 8, 9, and 10. When you compile it using gcc
2. What is the effect of the option *–Wall*?

**The ‘-Wall’ option enabled all warnings about constructions and language-specific warnings. In the case of the “Hello World” program I created, it specifically warned about the four unused variables I had in the program (‘tChar’, ‘fNum2’, ‘Fnum1’, and ‘dNum’). File name: Snyder\_HW\_01\_04\_a.txt**

1. What is the effect of the option *–pedantic*?

**The ‘-pedantic’ option will issue all warnings demanded by strict ISO C and C++; rejecting all programs not meeting the ISO C or ISO C++ or using forbidden extensions. This makes the code easier to compile with other compilers that implement the same standard. In the case of the “Hello World” program I created, no warnings were issued. File name: Snyder\_HW\_01\_04\_b.txt**

1. What is the difference in the size of the code compiled with and without *–g*?

**The size of the compiled code with the ‘-g’ option is larger than the code compiled without the ‘-g’ option. In the case of the “Hello World” program I created, the size of the code compiled without the ‘-g’ option was 7217 and with the ‘-g’ options was 8329. File name: Snyder\_HW\_01\_04\_c.txt**

1. What is the difference in the size of the code compiled with *–O1*, *–O2*, and *–O3*?

**The size of the code compiled will be optimized with the ‘-O1’ option, will be optimized even more with the ‘-O2’ option, and again even more than the other two options with the ‘O3’ option. In the case of the “Hello World” program I created, the size of the code compiled with the ‘-O1’ option was 7217, with the ‘-O2’ option was 7217, and with the ‘-O3’ option was 7217. Normally the size of the program decrease from –O1 to –O2 to –O3; however, since the “Hello World” program is very simple, I believe that is why I did not see a difference in size. File name: Snyder\_HW\_01\_04\_d.txt**

1. Open the C program in the debugger *gdb*. Set a breakpoint at the first line. Run the program to that point.
2. What is the current *ESP*?

**esp 0xbffff320 0xbffff320**

1. What is the current *EBP*?

**ebp 0xffff358 0xffff358**

1. What is the current *EIP*?

**eip 0x804843d 0x804843d <main+9>**

1. Use the *x* command to show the assembly language code for the next few commands to be executed.

**=> 0x804843d <main+9>: mov %gs:0x14,%eax**

**0x8048443 <main+15>: mov %eax,0x2c(%esp)**

**0x8048447 <main+19>: xor %eax,%eax**

**0x8048449 <main+21>: movl $0x0,0x18(%esp)**

**0x8048451 <main+29>: movb $0x21,0x1f(%esp)**

1. Use the *strace* command to list all of the system calls that your program makes.

**In the case of the “Hello World” program, the unique system calls used are execve, brk, access, mmap2, open, fstat64, close, read, set\_thread\_area, mprotect, munmap, write, and exit\_group. Many of the system calls appear multiple times. File name: Snyder\_HW\_01\_05.txt**

1. Use *objdump* to find the address of the following sections in your program:
2. *.text*

**Idx Name Size VMA LMA File off Algn**

**0 .text 0000005e 00000000 00000000 00000034 2\*\*2**

1. *.bss*

**Idx Name Size VMA LMA File off Algn**

**2 .bss 00000000 00000000 00000000 00000094 2\*\*2**

1. *.data*

**Idx Name Size VMA LMA File off Algn**

**1 .data 00000000 00000000 00000000 00000094 2\*\*2**

**File name: Snyder\_HW\_01\_09.txt**

1. Rewrite the “*helloASM.*asm” program described in class. Compile it and link it.
2. Include the source, object, and executable code.

**File names: Snyder\_HW\_01\_10.png; Snyder\_HW\_01\_10.asm; Snyder\_HW\_01\_10.o; executable excluded per e-mail directions**

1. Debug the program. What memory addresses contain the program’s environment strings?

**There are no environment strings. “Locals at unknown address”**

1. Debug the program. What memory addresses contain the program’s arguments?

**There are no program arguments. “Arglist at unknown address.”**

1. Debug the program, and stop it immediately before the syscall for *write*.
   * What is the current *ESP*?

**0xbfffec30: and %bl,-0x4(%edx)**

* + What is the current *EBP*?

**0xb7fc5a20 <\_IO\_2\_1\_stdout\_>: test %ch,(%edx)**

* + What is the current *EIP*?

**=> 0xb7fdd424 <\_\_kernel\_vsyscall+16>: pop %ebp**

1. Debug the program. Where is the message string located?

**$1 = 0x804a014**

1. What is the return value for the *write* syscall? Where is it located?

**Before syscall executes:**

**eax 0xffffffda -38**

**After syscall executes:**

**eax 0xe 14**

1. Write a C program that contains a stack based buffer overflow. Explain in detail why the program has a stack based buffer overflow flaw. Demonstrate the flaw by causing the program to crash with a segmentation fault. Include the state of the stack before the crash, and determine exactly why the program crashed.

**The program does not check the size of the data to be copied into the fixed sized buffer before it does the copy. Having more data than the buffer allows will cause the memory after the end of the fixed size buffer to be overwritten and a segmentation fault. File names: Snyder\_HW\_01\_11.c; Snyder\_HW\_01\_11.png**

1. Write a C program that contains a stack based buffer overflow, and make it SUID root. Run the program outside of the debugger as an unprivileged user, and exploit the overflow to obtain a root shell. Include a description of exactly how the program was exploited, and a screen shot showing a root shell was obtained.

**A vulnerable strcpy command reads in an arbitrary length character array from a file and stores it in a fixed length character buffer of size 24. Snyder\_HW\_01\_12\_exploit.c was developed to allow us to give root access within a shell through the Snyder\_HW\_01\_12\_call\_shellcode.c program that had already been loaded in memory. By debugging using GDB and ‘x/x128x buffer’, the address of str\_main in Snyder\_HW\_01\_12\_stack.c was obtained. This address was used to overflow the buffer for 64 bytes. In this case, 64 bytes worked because the distance between the “buffer” and the return address of the Snyder\_HW\_01\_12\_call\_shellcode in memory was less than 64 bytes. Several NOP operations were added to create a NOP sled for the new return address to point. At the end of the NOP sled, instructions to call the method used to open a shell were placed. The rest of the file used to overflow the buffer was filled with enough NOP bytes to overflow, in this case 256 bytes.**

**Steps:**

1. **As root, disable ASLR:**

**sysctl –w kernel.randomize\_va\_space=0**

1. **As root, compile the Snyder\_HW\_01\_12\_call\_shellcode.c program:**

**gcc –fno-stack-protector –z execstack –g –o Snyder\_HW\_01\_12\_call\_shellcode Snyder\_HW\_01\_12\_call\_shellcode.c**

1. **As root, compile the Snyder\_HW\_01\_12\_stack.c program with the stack check disabled:**

**gcc –fno-stack-protector –z execstack –g –o Snyder\_HW\_01\_12\_stack Snyder\_HW\_01\_12\_stack.c**

1. **As root, change the permissions on the resulting executable file:**

**chmod 4755 Snyder\_HW\_01\_12\_stack**

1. **As a regular user, compile the Snyder\_HW\_01\_12\_exploit program:**

**gcc –o Snyder\_HW\_01\_12\_exploit Snyder\_HW\_01\_12\_exploit.c**

1. **As a regular user, run the exploit program**

**./Snyder\_HW\_01\_12\_exploit**

1. **As a regular user, run the stack program**

**./Snyder\_HW\_01\_12\_stack**

1. **The result is a shell being opened with root privileges.**

**whoami (should return the ‘root’ user as shown)**

**File names: Snyder\_HW\_01\_12\_exploit.c; Snyder\_HW\_01\_12\_call\_shellcode.c; Snyder\_HW\_01\_12\_stack.c; Snyder\_HW\_01\_12.png**