Before starting the exam, you can fill out your name and other information of this page, but don’t open the exam until you are directed to start. Don’t put any of your answers on this page.

This exam contains 7 pages (including this cover page) and 4 questions. Once we tell you to start, please check that no pages are missing.

You may use any textbooks, notes, or printouts you wish during the exam, but you may not use any electronic devices: no calculators, smart phones, laptops, etc.

You may ask clarifying questions of the instructor or TAs, but no communication with other students is allowed during the exam.

Please read all questions carefully before answering them. Remember that we can only grade what you write on the exam, so it’s in your interest to show your work and explain your thinking.

By signing below you certify that you agree to follow the rules of the exam, and that the answers on this exam are your own work only.

The exam will end promptly at 12:30pm. Good luck!

Your name (print): ____________________________________________________________

Your UMN email/X.500: ________________________________________________________@umn.edu

Number of rows ahead of you: _________ Number of seats to your left, to an aisle: _________

Sign and date: ________________________________________________________________

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
1. (30 points) Matching definitions and concepts. Fill in each blank with the letter of the corresponding answer. Each answer is used exactly once.

(a) __O__ Intel’s name for a bit implementing $W \oplus X$

(b) __C__ Roughly a synonym of $W \oplus X$

(c) __A__ Choosing a random base address for memory regions

(d) __F__ A technical change to decrease the possibility of attack

(e) __H__ A safe place to store return addresses

(f) __I__ Falsifying your identity in communication

(g) __J__ A C library routine that executes a shell command

(h) __D__ An amount of randomness measured in bits

(i) __M__ Represented with a dashed rectangle

(j) __G__ A code reuse attack using complete functions

(k) __K__ Modifying information that should be protected

(l) __B__ Property of information protected from disclosure

(m) __E__ A Unix system call to switch to a new program

(n) __N__ A Windows system call to change memory permissions

(o) __L__ A value that can’t be copied because it signifies the end

2. (16 points) Stack buffer overflow, in source code.

The two C functions src1 and src2 both implement similar functionality, but the different order in which they do certain operations has a significant effect. Assume that the argument s to both functions is non-null, but could point to any characters. One of the functions is safe, in the sense that it will never invoke undefined behavior. But the other function is unsafe: for some inputs, it will invoke undefined behavior. Depending on how it is compiled, this means it could crash or allow an attack.

The functions use subroutines named strlen_nl and strcpy_nl, which are similar to the standard library functions with similar names, but use a newline character (’\n’, hex 0x0a) as a terminator instead of a null character.

```c
size_t strlen_nl(const char *s) {
    size_t count = 0;
    while (*s != '\n') {count++; s++;} return count;
}

char *strcpy_nl(char *dst, const char *src){
    char *p = dst; const char *q = src;
    while (*q != '\n') { *p++ = *q++; }
    *p++ = '\n';
    return dst;
}

int src1(char *s) {
    char buf[16];
    size_t len;
    len = strlen_nl(s);
    if (len >= 16) {
        puts("Input too long!");
        exit(1);
    }
    strcpy_nl(buf, s);
    return buf[0];
}

int src2(char *s) {
    char buf[16];
    size_t len;
    len = strlen_nl(s);
    strcpy_nl(buf, s);
    if (len >= 16) {
        puts("Input too long!");
        exit(1);
    }
    return buf[0];
}
```

(a) The buffer buf can hold 16 characters. Why is it nonetheless a good idea that the code that checks for the input string being too long uses the condition len ≥ 16 (equivalent to len > 15), rather than len > 16?

Like its non-nl namesake, strlen_nl does not count the terminating character (here a newline) as part of the length. But the terminating character will be written to the buffer by strcpy_nl. So it would be unsafe to call strcpy_nl when the length is 16; it would overflow the buffer by one byte.

(b) Between src1 and src2, which one is safe and which one is unsafe? Briefly explain why.

The difference between the functions is the relative ordering of the length check and the call to strcpy_nl: in src1 the check comes before the copy, and in src2 the copy comes first. Intuitively, you also want to check the safety of an operation before you perform it: if you find out that an operation was unsafe after you performed it, something bad may have already happened. According to the rules of C in particular, overflowing a buffer leads to undefined behavior, so it is not safe to assume anything about the behavior of src2 after the strcpy_nl call. Only src1 is safe.
3. (26 points) Stack buffer overflow, in machine code.

Below are four function definitions in Linux/x86-64 assembly code, compiled from src1 and src2 from the previous question. Two of the compilations come from each of the two source functions, but with different compiler options; the labels A through D were assigned randomly. The code that handles the error case is always the same, so we’ve separated it out with the label error_handler. Only one of these four versions is vulnerable to a stack buffer overflow attack overwriting its return address.

```
A: push %rbx
    sub $0x10,%rsp
    mov %rdi,%rbx
    call strlen_nl
    cmpq $0xf,%rax
    ja error_handler
    mov %rsp,%rdi
    mov %rbx,%rsi
    call strcpy_nl
    movsbl (%rsp),%eax
    add $0x10,%rsp
    pop %rbx
    ret

B: push %rbp
    push %rbx
    sub $0x18,%rsp
    mov %rdi,%rbx
    call strlen_nl
    mov %rax,%rbp
    mov %rsp,%rdi
    mov %rbx,%rsi
    call strcpy_nl
    cmpq $0xf,%rbp
    ja error_handler
    movsbl (%rsp),%eax
    add $0x18,%rsp
    pop %rbx
    pop %rbp
    ret

C: push %rbp
    mov %rsp,%rbp
    sub $0x30,%rsp
    mov %rdi,-0x28(%rbp)
    mov -0x28(%rbp),%rax
    mov %rax,%rdi
    call strlen_nl
    mov %rax,-0x8(%rbp)
    mov -0x28(%rbp),%rdx
    lea -0x20(%rbp),%rax
    mov %rdx,%rsi
    mov %rax,%rdi
    call strcpy_nl
    cmpq $0xf,-0x8(%rbp)
    ja error_handler
    movzbl -0x20(%rbp),%eax
    movsbl %al,%eax
    mov %rbp,%rsp
    pop %rbp
    ret

D: push %rbp
    mov %rsp,%rbp
    sub $0x30,%rsp
    mov %rdi,-0x28(%rbp)
    mov -0x28(%rbp),%rax
    mov %rax,%rdi
    call strlen_nl
    mov %rax,-0x8(%rbp)
    mov -0x28(%rbp),%rdx
    lea -0x20(%rbp),%rax
    mov %rdx,%rsi
    mov %rax,%rdi
    call strcpy_nl
    cmpq $0xf,-0x8(%rbp)
    ja error_handler
    movzbl -0x20(%rbp),%eax
    movsbl %al,%eax
    mov %rbp,%rsp
    pop %rbp
    ret
```

message:
    .string "Input too long!"

error_handler:
    mov $message,%rdi
    call puts
    mov $0x1,%edi
    call exit
Here is an example of an input, in the format of a C string, that would overwrite the return address of the function with the value 0x4012e2 if it is given as the argument to the vulnerable version:

"AAAAAAAABBBBBBBBxxxxxxxx\0\0\0\0\0\0\0\0\0\0yyyyyyy\xe2\x12\x40\0\0\0\0\n"

(a) Write the letters of the two versions compiled from src1: ___A___ ___C___

(b) Write the letters of the two versions compiled from src2: ___B___ ___D___

(c) For each of the versions, which location(s) hold the value of the variable len? For each version, write one or more locations, where each location is either a register (e.g., %rcx), or a stack location indicated as an offset from the location a register points to (e.g., 42(%rcx) represents the location 42 bytes beyond where the register %rcx points).

A: %rax
B: %rbp
C: -8(%rbp)
D: -8(%rbp)

(d) Write the letter of the version that is vulnerable: ___D___

(e) Briefly explain why this and only this version is vulnerable: In version D only, the buffer overflow in strcpy nl can change the value of len in a way that will keep the length-check branch from working correctly. A and B are safe from the attack because they store the length in a register, which can never be affected by a buffer overflow. C is safe because the length check comes before the copy. But in D, the overflow can change the stack location holding len to a small value that will pass the length check (1 in the sample attack), even after copying too many bytes and overwriting the return address.

Here are some reminders about Linux/x86-64 assembly language. We use “AT&T” syntax, which means that the operand that is modified in an instruction always comes last, even though that means that subtraction (sub) and comparison are backwards from normal math. The cmp instruction compares two values, and the suffix q indicates that it operates on 64-bit values. The conditional jump instruction ja transfers control to operand label if the result of a previous comparison was greater-than (“above”) according to unsigned arithmetic. The instruction lea computes an address or other numeric value using addressing-mode operations. The mov instruction copies data from its first operand to its second; the sbl and zbl variants expand from an 8-bit source to a 32-bit destination with sign extension or zero-extension respectively. push allocates 8 bytes by decreasing the stack pointer %rsp and copies a value the stack, while pop copies a value from the stack and increments the stack pointer by 8 bytes. The first two arguments to a function are passed in registers %rdi and %rsi, and a return value is in the register %rax. The function exit terminates the program.
4. (28 points) Multiple choice. Each question has only one correct answer: circle its letter.

(a) All of the following printf format specifiers might sometimes produce only a single byte of output, except:
   A. %ld  B. %c  C. %s  D. %d  E. %100d
%c always produces a single character (byte) of output. %ld and %d can both do so if their argument is between 0 and 9. %s can produce a single byte of output if its argument string is one byte long before the null terminator. But %100d will always print 100 bytes of output, because it will pad its integer output with spaces. (When int is 32 bits as it now usually is, the integer itself could only take up to 11 characters in signed decimal, such as in -2147483648.)

(b) If x is a 32-bit signed integer (like an int), all of the following operations could overflow, except:
   A. x - 1  B. x / 2  C. x + 1  D. x * 2  E. x + x + x
Adding any positive value, or multiplying by 2 or 3, could all overflow when their result would be greater than or equal to $2^{31}$. Subtracting a positive value can overflow when its result is less than $-2^{31}$. But dividing a number by two always decreases its absolute value (or leaves it at 0), so it can't overflow.

(c) Suppose that an array field within a struct allocated with malloc can be overflowed via strcpy. All of the following might be overwritten except:
   A. an integer field later in the structure
   B. a return address
   C. a pointer field in a different heap-allocated object
   D. heap metadata for the allocation containing the overflow
   E. metadata for another heap allocation
Return addresses are always stored on the stack, whereas objects allocated with malloc are stored on the heap. The stack and the heap are separate memory regions with a large un-allocated gap between them, so a sequential overflow could never overwrite from the heap to the stack. But all the other answers are things stored on the heap which might be overwritten.

(d) Addresses on x86-64 are stored in 64 bits, but current systems don’t use all 64. In one common configuration, the top 17 bits of an address are required to all be the same, and if these bits are all 1, the address is reserved for the OS kernel. Also, pages are 4096 bytes long, and keeping memory regions page-aligned is important for performance. If these were the only relevant restrictions, the number of locations that could be chosen for one user-space memory region in ASLR is:
   A. 2^{12}  B. 2^{20}  C. 2^{34}  D. 2^{35}  E. 2^{36}
An equivalent way of stating the restrictions is that the a location used as the starting address of a memory region needs to have its top 17 bits and low 12 bits all zero. This leaves $64 - 17 - 12 = 35$ bits in between that can be either 0 or 1 in any combination.
(e) Arguably one of the most important features of pen-and-ink signatures in the physical world is that you can confront someone later with a document they have signed, and it is hard for them to deny having signed it. In our terminology, the property being provided here is:
A. integrity  B. non-repudiation  C. availability  D. confidentiality  E. invariance

(f) Suppose your company is considering switching to two-factor authentication (similar to UMN’s use of Duo) with a service provided by an outside company named AuthCorp, and your considering threats that might arise from AuthCorp. When logging in, your users will both provide a password checked on your company’s service, and be authenticated via AuthCorp’s app. Both the password and using the app are required, so even if AuthCorp is malicious, as long as they don’t know users’ other passwords, this threat class is mitigated:
A. spoofing  B. tampering  C. repudiation  D. information disclosure  E. denial of service

(g) On the other hand, because AuthCorp’s service must be working correctly for users to log in, AuthCorp might still be a source of this threat class:
A. spoofing  B. tampering  C. repudiation  D. denial of service  E. escalation of privilege

For instance, a malicious AuthCorp could reject all login attempts.