Computer Science 4271
Fall 2022
Midterm 1 (fixed, solutions)
October 18th, 2022
Time Limit: 75 minutes, 4:00pm-5:15pm

- 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 8 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 5:15pm. Good luck!

Your name (print): _____

Your UMN email/X.500: _____`@umn.edu`

Number of rows ahead of you: _____ Number of seats to your left: _____

Sign and date: _____

Question	Points	Score
1	20	
2	24	
3	28	
4	28	
Total:	100	

1. (20 points) Matching definitions and concepts. Fill in each blank with the letter of the corresponding answer. Each answer is used exactly once.

   (a) __**B**__ A sequence of instructions ending in a return

   (b) __**H**__ Falsely denying that an action took place

   (c) __**G**__ A bit used by AMD to implement W $\oplus$ X

   (d) __**C**__ Freedom from unauthorized data modification

   (e) __**J**__ A defense that limits attackers to code reuse

   (f) __**E**__ A function to change memory permissions

   (g) __**D**__ A function to copy a given number of bytes

   (h) __**F**__ Padding code for shellcode

   (i) __**I**__ A function to copy bytes up to a null terminator

   (j) __**A**__ Choosing random base addresses for memory regions

   A. ASLR B. gadget C. integrity D. `memcpy` E. `mprotect` F. NOP sled G. NX H. repudiation I. `strcpy` J. W $\oplus$ X

Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
0	00	NUL	16	10	DLE	32	20		48	30	0	64	40	@	80	50	P	96	60	‘	112	70	p
1	01	SOH	17	11	DC1	33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
2	02	STX	18	12	DC2	34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
3	03	ETX	19	13	DC3	35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
4	04	EOT	20	14	DC4	36	24	$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
5	05	ENQ	21	15	NAK	37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
6	06	ACK	22	16	SYN	38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
7	07	BEL	23	17	ETB	39	27	’	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
8	08	BS	24	18	CAN	40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
9	09	HT	25	19	EM	41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
10	0A	LF	26	1A	SUB	42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
11	0B	VT	27	1B	ESC	43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k	123	7B	{
12	0C	FF	28	1C	FS	44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	7C	|
13	0D	CR	29	1D	GS	45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m	125	7D	}
14	0E	SO	30	1E	RS	46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
15	0F	SI	31	1F	US	47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

2. (24 points) STRIDE classification.

   In each of the following scenarios, we describe 6 threats, one each from the STRIDE classification of spoofing, tampering, repudiation, information disclosure, denial of service, and escalation of privilege. Write the letters S, T, R, I, D, and E in the appropriate order in the blanks according to which type of threat each is. In our answer, each of the letters is used exactly once in each scenario.

   Optionally, there is also one blank next to a blank space for each scenario. If you don't like our examples, you can write one new threat and STRIDE classification of your own in this space, and if it's a good example, it can compensate for one other threat in the scenario we marked wrong.

   In each scenario, people whose names start with A are attackers, and those whose names start with V are victims.

   (a) In-person voting on election day. Most of these can work whether the voting is on paper or electronic.

   __**R**__ Alice votes once, comes back later, and votes again claiming it is her first time

   __**D**__ Alice pulls the fire alarm and the polling place is evacuated

   __**T**__ Alice changes Vicki's mayoral vote from Bob to Charlie

   __**E**__ Alice is a regular voter, but gets the election judge's keyring

   __**I**__ Alice gets a list of everyone who voted for Bob for mayor

   __**S**__ Alice uses a fake ID to cast a ballot under Vicki's name

   ___

   (b) Oliver's online olive oil electronic commerce website. Adam is a customer, Alex is a competitor, and Arnold is just a vandal.

   __**E**__ Arnold discovers the configuration page `admin.php` is not password protected

   __**D**__ Alex files a trademark lawsuit to get Oliver's web hosting taken down

   __**S**__ Adam orders rancid olive oil to be delivered to Victor

   __**R**__ Adam gets a delivery, but claims it was lost and asks for a refund

   __**I**__ Adam gets Victor's credit card number

   __**T**__ Arnold changes the product descriptions to add awful puns

   ___

3. (28 points) Multiplication and memory allocation.

   Consider the following C function which attempts to allocate memory for, and then read in, a number of integers controlled by the argument `num_ints`. Use it to answer the questions on the following page.

```
int *alloc_and_read(unsigned char num_ints) {
    unsigned char size = sizeof(int) * num_ints;
    if (size < num_ints) { /* overflow check */
        fprintf(stderr, "Uh-oh, overflow!\n");
        exit(1);
    }
    int *ary = malloc(size);
    if (!ary) {
        fprintf(stderr, "Allocation failed\n");
        exit(1);
    }
    int i;
    for (i = 0; i < num_ints; i++)
        ary[i] = read_int();
    return ary;
}
```

   Assume that `sizeof(int)` is 4, as it is on x86-64. We'll use the variable n to represent the value of `num_ints`, which is between 0 and 255 in decimal (`0x00` to `0xff` in hex). Because the variable `size` is also only an `unsigned char`, its value is also limited to between 0 and 255. Specifically, the value stored in `size` will be $(4 \cdot n) \bmod 256$. The "mod 256" operation is also the same as discarding all but the two lowest hex digits, or all but the 8 lowest bits, of a number.

   You can use decimal, hexadecimal, or binary in your answers, but to keep them distinct, write hexadecimal numbers with a `0x` prefix and binary numbers with an `0b` prefix. It is enough to write just the formula or number if it is correct, but a short explanation of your answer may help us give partial credit. You don't need to simplify formulas.

   This question is a variation on the idea of integer overflow on multiplication that can lead to a buffer overflow, something we talked about it class and that came up in lab 2. The key point of this question is that the code in this function that describes itself as an overflow check is incomplete: there are some overflow conditions that are not caught be the check, so the code is still vulnerable. Multiplication by 4 is supposed to make any positive number only larger, so multiplying a non-negative value by 4 and getting a result that is smaller could only be the result of an overflow. You might have previously seen addition of two numbers, or multiplication by 2, checked for overflow in this way. However for multiplication by 4 getting a smaller result is not a complete check: a smaller result means you had overflow, but you can also have overflow without the result being smaller. The parts of the question were intended to lead you through this, by examining the condition on the `if` *and the actual circumstances of overflow, and seeing the difference between them.*

   There are three important quantities that come up during execution here, which the notation in the question suggests you refer to as n*,* $4n$*, and* $4n \bmod 256$*. We always have* $0 \leq n \leq 4n$*, with*

the inequalities being strict if n in non-zero. ($n = 0$ is a special case because it is unspecified whether `malloc` *will succeed or fail.) But $4n \bmod 256$ can be anywhere between 0 and $4n$.*

We wanted you to answer parts (a) and (b) by writing down a formula comparing two values in the most direct way, without trying to simplify the formula or solve it as an explicit set of values. Though solving the formulas in (a) and (b) could potentially make part (c) easier, it was easy to make mistakes in doing this.

(a) Write a mathematical formula, in terms of the variable n, which will be true for those values of n that cause the message "Uh-oh, overflow!" to be printed.

$\boxed{4n \bmod 256 < n}$

You can get this formula by just substituting the definitions of `size` *and* `num_ints` *into the if condition that guards the message printing.*

Finding the explicit ranges of n that satisfy this formula requires more computation, which you would have been better off not trying to do during the exam. But if you're curious: the values of n for which this relationship hold come in three disjoint ranges. $4n \bmod 256 = 0$ whenever n is a multiple of 64, so for $n = 64$, $n = 128$, and $n = 192$, the result is 0 and clearly less than n. The inequality will still hold for values slight larger than all of these: for instance for $n = 70$, $4n \bmod 256 = (4 \cdot 70) \bmod 256 = 24$ and $24 < 70$. As for when the ranges end, you can use the intuition that it should be about at those values of n where $4n \bmod 256 = n$, i.e. $4n \pm k \cdot 256 = n$ for some integer i. Besides the obvious cases of $n = 0$ and $n = 256$ (which is one past the largest possible value), the other cutoffs should be around solutions of $k \cdot 256 = 3n$, so $3n = 256$ and $3n = 512$. Of course these don't come out to round numbers: $256/3$ is $85 + 1/3$ and $512/3$ is $170 + 2/3$. So you'd round the threshold values down to get the last value for which the inequality holds. Thus the ranges are [64, 85], [128, 170], and [192, 255].

(b) Pretend for a moment that the `if` statement labeled "overflow check" were not present. Write a mathematical formula, in terms of the variable n, which will be true for those values of n where the function will write beyond the area of memory allocated for `ary`.

$\boxed{4n \bmod 256 < 4n}$

$4n \bmod 256$*, as before, is* `size` *and the number of bytes that will be allocated. $4n$ is the number of bytes that will be written: each iteration of the loop reads 4 bytes, and the loop executes n times, so it's like multiplication without overflow.*

This formula has a simpler solution than the one in part (a): it's equivalent to $n \geq 64$. That's because in this case the result of the overflow doesn't make a difference: any overflow will cause the result to be less than it should be. $n = 63$ is the largest value that does not overflow, while $n = 64$ is the smallest value that does overflow, and all values greater than 64 also overflow. If you think of multiplying by four as equivalent to shifting left by two in binary, the top two bits of n will be the 9th and 10th bits after the shift if there is no size limit, but if you shift within 8 bits these values are lost and it's as if the 9th and 10th bits are always 0. So there will be overflow any time the top two bits of n are not both 0.

(c) Give one specific value for n that will cause the function with the overflow check to write beyond the area of memory allocated for `ary`. This will need to be a value for which the formula in part (b) is true, while the formula in part (a) is false; this also implies that those formulas should be different.

Many possible answers. If you compared the solved versions of the formulas from (a) and (b), the ranges of values covered by (b) but excluded by (a) will be [86, 127] and [171, 191].

However since we only asked for a single value here there are easier things to do.

The most common correct answer people picked by guess and check was 100, probably since it's a number that's easy to work with in decimal but significantly bigger than 64. $4 \cdot 100 = 400$*,* $400 \bmod 256 = 400 - 256 = 200 - 56 = 150 - 6 = 144$*, and* $144 > 100$*.*

Another approach is to think about what's happening in binary after a left shift by two. If the original value n *is of the form* `0bAABBCCDD`*, then* $4n \bmod 256$ *will be* `0bBBCCDD00`*. We know that* `AA` *can't be* `00` *for there to be overflow, but the easiest way to make* `0bBBCCDD00` *greater than* `0bAABBCCDD` *is to have* `BB` *bigger than* `AA`*. So you can set* `AA` *to* `01` *and* `BB` *to* `10`*, for instance* $n =$`0b01100000`*. This is* `0x60`$=96$*. But by similar reasoning any value whose first hex digit is* `0x6` *(*`0b0110`*),* `0x7` *(*`0b0111`*), or* `0xb` *(*`0b1011`*) will work. Or you could have* `AA`$=$`BB` *as long as* `CCDD` $>$ `BBCC`*, which accounts for* `0x56`–`0x5f` *and* `0xab`–`0xaf`*.*

4. (28 points) Overwriting an address.

   The following function from a Linux/x86-64 program has a buffer overflow vulnerability. Depending on the contents of the string `attack`, which we assume is under the control of an attacker, the return address of the function `func` might be overwritten. The program is compiled without PIE or stack canaries.

   Below are excerpts of the relevant code in C and assembly language.

```
void func(char *attack) {
    char buf[8];
    strcpy(buf, attack);
}
```

```
1:    sub    $0x10, %rsp
2:    mov    %rdi, %rsi
3:    lea    0x8(%rsp), %rdi
4:    call   strcpy
5:    add    $0x10, %rsp
6:    ret
```

   The normal return address of the function is `0x4011c0`. Assume that in order to start a code reuse attack, the attacker wants to change the return address to `0x401171`.

   In the left column, below, are 7 possible contents for the string `attack` passed to the function, written using the same rules as for string constants in C. A sequence of `\x` followed by two hex digits represents a single character (byte) whose numeric value is given by the following two hex digits. For instance `\x2a` represents the byte with value `0x2a`, decimal 42.

   In the right column are 5 different numeric values for the return address at the time when the function returns. Write the letter of an entry in the right column in the blank on the left to match an attempted attack with the effect it has on the return address. Each answer might be used once, more than once, or not at all.

   The tricky part of this question, both in intent and as it worked out, was null terminators. Recall that `strcpy` *doesn't take the length of a string to copy as an argument: instead, its source argument should be a null-terminated string, and it copies just as many bytes as there are up to and including the null terminator. This flexibility is convenient for attackers creating buffer overflows, but the behavior of stopping at a null terminator is a limitation to what data an overflow can write. You may recall this limitation is also part of the terminator canary idea.*

   From the source code you can see that any string of length more than 7 is in danger of overflowing the buffer: 7 content characters, plus the null terminator, are the maximum that the buffer can safely hold. However the source code doesn't tell you what will happen beyond that, since information about the memory layout is only present in the machine code. The subtraction instruction 1 allocates 16 bytes on the stack, but the `lea` *instruction 3 that constructs the buffer pointer to pass to* `strcpy` *adds 8 bytes to the starting address of that 16-byte region, so there is no padding: overflowing the buffer by even one byte can affect the return address. You could have also guessed that this was the layout based on the provided answers.*

   There is also a post on Piazza with C code for a program to reproduce the results of this question, if you're curious.

A. `0x4747474700401171`

B. `0x0000000000400042`

C. `0x0000000000400071`

D. `0x00000000004011c0`

E. `0x0000000000401171`

(a) __**D**__ `AAAA`

This is an example of the no-overflow case: the return address is unchanged.

(b) __**D**__ `AAAAAAA`

This is another no-overflow case. Seven A characters plus the terminating null character still fit in the 8-byte buffer.

(c) __**B**__ `AAAAAAAAB`

An input with 9 content characters leads to a total overwrite of two bytes: the last byte in the attacker-controlled input, and then a null terminator. So the least-significant byte of the return address changes from `0xc0` *to* `0x42` *(the ASCII code for B), and the second-least-significant byte changes from* `0x11` *to 0x00. If you were looking for an answer of* `0x0000000000041142` *for this, you were forgetting the null terminator.*

(d) __**C**__ `AAAAAAAA\x71`

This part is exactly analogous to the previous part, except that a different value overwrites the least-significant byte.

(e) __**E**__ `AAAAAAAA\x71\x11\x40`

This is an example of a successful overwrite: the attacker replaces the low three bytes of the return address with byte values of their choosing, and then the null terminator byte harmlessly overwrites a byte that was already 0.

(f) __**E**__ `AAAAAAAAq\x11@`

This input works the same as the one in the previous part because it is really the same input: `q` *and* `@` *are just the ASCII characters with numeric values* `0x71` *and* `0x40` *respectively.*

(g) __**E**__ `AAAAAAAA\x71\x11\x40\x00GGGG`

This was the trickiest of the parts, because it has a very appealing-looking incorrect answer. The input is the same as the successful attack, but with a null byte and 4 capital G characters added on to the end. That makes it look very similar to answer A, which has the successful result E with four of the most-significant bytes changed to `0x47`*, which is indeed the ASCII code for G. It might also have been tempting to use this answer if you noticed that all the other answers were used, though we tried to disclaim the assumption that every answer would be used. But that appealing answer is not correct, because the null byte causes the copying to stop. So instead this input is again equivalent to the two above.*