Full Name:______________________________________

Midterm Exam, Fall 2016 Date: Nov 2nd, 2016

Instructions:

- This midterm exam takes 70 minutes. Read through all the problems and complete the easy ones first.
- This exam is OPEN BOOK. You may use any books or notes you like. However, the use of any electronic devices including laptops, ipads, phones etc. is forbidden.

1 (xx/25)	2 (xx/25)	3 (xx/25)	4 (xx/25)	Total (xx/100)

1 Multiple choice questions (25 points):

Answer the following multiple-choice questions. Circle *all* answers that apply. Each problem is worth 5 points.

A. After executing the assembly instruction `movl $0x12345678, %eax` on a x86-64 Intel CPU, what is the content of register %al? Hint: Register %al represents the lower 8-bit of the 32-bit register %eax.

1. `0x12`
2. `0x34`
3. `0x56`
4. `0x78`
5. None of the above

B. Suppose variable `b` is of `unsigned byte` type, which of the following statement sets the i-th most-significant bit of `b` to 1 and leave the rest of the bits unchanged? (We assume i's range is $[0, 7]$ and we refer to the most significant bit of a byte as its 0-th most-significant bit)

1. `b | = ((1<<i)>>i)`
2. `b | = (1<<(7-i))`
3. `b | = (1<<(8-i))`
4. `b & = ((1<<i)>>i)`
5. `b & = (1<<(7-i))`
6. `b & = (1<<(8-i))`
7. None of the above

C. Suppose `y` is an unsigned byte of any value, which of the following equality always holds? (Note that $\wedge$ is the C operator for bitwise XOR and $\sim$ is the C operator for bitwise NOT)

1. (y | 0x00) == y
2. y & 0xff) == y
3. (y $\wedge$ y) == 0x00
4. (y $\wedge$ $\sim$ y) == 0xff
5. (y | (x | y)) == x

D. Consider the following code snippet,

```
char *names[3] = {"alice", "bob", "claire"]
char **p;
p = names;
p++;
```

After executing the above, what is the value of `p[0][2]`?

1. 'i'
2. 'b'
3. 'a'
4. 0
5. Undefined
6. None of the above

E. Consider the following piece of code,

```
#include <stdio.h>
float f = 0.2;
float bound = 1.0e5;
int i = 0;
while ( f < bound) {
   f += 0.2;
   i++;
}
```

Which is the value of `i` after executing the above snippet?

1. 500000
2. 100000
3. Some value close to (but not identical to) 500000, e.g. 498256
4. None of the above

2 C basics (25 points):

Ben Bitddle wants to encode a binary array into a C string in hexadecimal format. Below is the skeleton of Ben's program.

```
char
Bin2Hex(unsigned char c)
{
...
}

void
EncodeToHex((unsigned char *)buf, int size, char *str)
{
...
}

void
TestHex() {
   unsigned char arr[5] = {1, 2, 3, 4, 255};
   char *str1;
   str1 = EncodeToHex(arr, sizeof(arr));
   printf("0x%s\n", str1); //should print 0x01020304ff

   int x = 1024;
   char *str2;
   str2 = EncodeToHex((unsigned char *)&x, sizeof(x));
   printf("0x%s\n", str2);

}
```

(a) (5 points) In the `TestHex` function, `str1` should be the C string "01020304ff", i.e., the expected first line of the printout should be `0x01020304ff`. What is the expected second line of the printout after executing the `TestHex` function?

(b) (10 points) Before implementing `EncodeToHex`, Ben decides to first implement a helper function called `Bin2Hex`. `Bin2Hex` takes a byte `c` as an argument and returns its corresponding ASCII character in hex. The caller of `Bin2Hex` must ensure that `c` has a valid range from 0 to 15, i.e. $c \in [0, 15]$. For example, if `c`=0, then `Bin2Hex` should return `'0'`. If `c`=10, then `Bin2Hex` should return `'a'`. Please complete `Bin2Hex` below.

```
char
Bin2Hex(unsigned char c)
{



}
```

(c) (10 points) Implement `EncodeToHex` using the `Bin2Hex` helper function. The `EncodeToHex` function takes two arguments. The first argument, `buf`, is the given byte array. The second argument, `size`, is the number of elements of the `buf` byte array. `EncodeToHex` should allocate a character buffer to store the encoded hex string and return the pointer to the allocated character buffer containing hex string. Note: you should ensure that the returned buffer contains a proper null-terminated C string.

```
char *
EncodeToHex((unsigned char *)buf, int size)
{
```

3 C and Buffer Overflow (25 points)

Ben Bitdiddle wants to parse a student grade file in the *.csv format. Each line of the file contains two fields, student name followed by his/her lab-1 grade. The two fields are separated by the character `';'`. Below are some example lines of the file.

```
John Smith;50
Emma Min;85
Larry Kim;85
```

(a) (10 points) Ben writes the function `ParseGrade`, which takes a line of text as input and extracts the first and second field of the line.

The first argument of `ParseGrade`, `input`, is a null-terminated C string that contains a line of the text file. The second argument, `name`, points to a character buffer. When `ParseGrade` returns, the character buffer pointed to by `name` should contain a proper null-terminated C string that is the parsed student name. The third argument, `grade`, points to an integer. When the function returns, the interger pointed to by `grade` should contain the parsed lab-1 grade of the student. If parsing is successful, the function returns 0. Otherwise, the function returns 1.

(see next page)

Please help Ben complete the `ParseGrade` function. You should use the `atoi` function in standard C library (see appendix I) to parse the integer. You may also optionally use the `strchr` function to locate a character in a given string (see appendix I). Note that if you want to use other functions from the standard C library, you must use them correctly.

```
int
ParseGrade(char *input, char *name, int *grade)
{




}
```

Ben write a program to parse a line of student grade (see code below). Function `ParseOneLineExample` reads one line from the terminal input using `getline` (whose implementation is not shown). `getline` internally malloc-s a buffer big enough to hold the line read and returns the address of the allocated buffer.

```
void ParseOneLineExample()
{
   char *input;
   char name[10];
   int grade;
   input = getline(); //read one line from standard input
   ParseGrade(input, name, &grade);
   return;
}

void MyGreatHack()
{
   printf("All students get As\n");
}

void main()
{
   ParseOneLineExample();
}
```

The corresponding dissembled code is shown as follows:

```
0000000000400651 <ParseGrade>:
  400651: 48 83 ec 08           sub    $0x8,%rsp
      .... (lines omitted)
  400670: c3                    retq

0000000000400696 <MyGreatHack>:
  400696: 48 83 ec 08           sub    $0x8,%rsp
  40069a: bf 64 07 40 00        mov    $0x400764,%edi
  40069f: e8 4c fe ff ff        callq  4004f0 <puts@plt>
  4006a4: 48 83 c4 08           add    $0x8,%rsp
  4006a8: c3                    retq

0000000000400671 <ParseOneLineExample>:
  400671: 48 83 ec 28           sub    $0x28,%rsp
  400675: b8 00 00 00 00        mov    $0x0,%eax
  40067a: e8 ae ff ff ff        callq  40062d <getline>
  40067f: 48 8d 54 24 0c        lea    0xc(%rsp),%rdx
  400684: 48 8d 74 24 10        lea    0x10(%rsp),%rsi
  400689: 48 89 c7              mov    %rax,%rdi
  40068c: e8 c0 ff ff ff        callq  400651 <ParseGrade>
  400691: 48 83 c4 28           add    $0x28,%rsp
  400695: c3                    retq

00000000004006a9 <main>:
  4006a9: 48 83 ec 08           sub    $0x8,%rsp
  4006ad: b8 00 00 00 00        mov    $0x0,%eax
  4006b2: e8 ba ff ff ff        callq  400671 <ParseOneLineExample>
  4006b7: 48 83 c4 08           add    $0x8,%rsp
  4006bb: c3                    retq
  4006bc: 0f 1f 40 00           nopl   0x0(%rax)
```

(b) (2 points) Based on the dissembled code, after the instruction at address `000000000040068c` has been executed, what's the next instruction to be executed? And where is that instruction stored?

(c) (2 points) In scenarios where there is no buffer overflow, after the instruction at address `0000000000400695` has been executed, what's the next instruction to be executed? And where is that instruction stored?

(d) (2 points) Before executing the first instruction in function `ParseOneLineExample`, suppose register %rsp has the value `0x7fffffffe528`, what's the 8-bytes stored at memory location starting at `0x7fffffffe528`?

(e) (2 points) Before executing the first instruction in function `ParseOneLineExample`, suppose register %rsp has the value `0x7fffffffe528`, what's starting address of 10-byte `name` array?

(f) (7 points) Suppose you are the attacker who wants to exploit the buffer overflow bug in `ParseOneLineExample` to hijack the control flow of the program to invoke the `MyGreatHack` function. How would you construct the input for the hack? Please be as concrete as possible.

4 Assembly (25 points):

Ben Bitdiddle is given the following code skeleton and their corresponding assembly code. (Question starts on the next page.)

```
void foo(                 )
{
}

void bar(                 )
{
}

int mystery()
{
   int a[3] = {1, 2, 3};
   int *b;
   b = NULL;
   foo(         );
   bar(                 );
   return *b;
}
```

The corresponding dissembled code is shown as follows:

```
foo:
   addq    $4, %rdi
   movq    %rdi, (%rsi)
   ret

bar:
   addl    $1, (%rdi)
   ret

mystery:
   subq    $0x28, %rsp
   movl    $0x1, 0x10(%rsp)
   movl    $0x2, 0x14(%rsp)
   movl    $0x3, 0x18(%rsp)
   movq    $0x0, 0x8(%rsp)
   leaq    0x8(%rsp), %rsi
   leaq    0x10(%rsp), %rdi
   call    foo
   movq    0x8(%rsp), %rdi
   call    bar
   movq    0x8(%rsp), %rax
   movl    (%rax), %eax
   addq    $0x28, %rsp
   ret
```

(a) (5 points) Suppose at the time of entering function `mystery` (before executing its first instruction `sub $0x28, %rsp`), register `%rsp` contains the value `0x7fffffffe528`. The first instruction `sub $0x28, %rsp` allocates 40 bytes on the stack to hold local variables. Note that the compiler typically generates code that allocates more space than is strictly needed.

We know that the local variables `a` and `b` are located somewhere on the stack. What's the starting address of the 12-byte memory region that holds the 3-integer array `a`? What is the starting address of the 8-byte memory region that holds the pointer variable `b`?

(b) (5 points) How many arguments does function `foo` take? What are their types? And what does function `foo` do? Answer these questions by giving the corresponding C code for function `foo`.

```
void foo(
{



}
```

(c) (5 points) How many arguments does function `bar` take? What are their types? And what does function `bar` do? Answer these questions by giving the corresponding C code for function `bar`.

```
void bar(
{









}
```

(d) (5 points) How does `mystery` calls functions `foo` and `bar`? Filling in the blanks at line 3 and 4.

```
int
mystery()
{
1:  int a[3] = {1, 2, 3};
2:  int *b;

3:  foo(                                          );

4:  bar(                                          );

5:  return *b;

}
```

(e) (5 points) What integer value does `mystery` return?

Appendix I: atoi

```
ATOI(3)                                                                                   Li

NAME
       atoi - convert a string to an integer

SYNOPSIS
       #include <stdlib.h>

       int atoi(const char *nptr);

DESCRIPTION
       The atoi() function converts the initial portion of the string pointed to by nptr to
       atoi() does not detect errors.

RETURN VALUE
       The converted value.


STRCHR(3)                        Linux Programmer's Manual

NAME
       strchr - locate character in string

SYNOPSIS
       #include <string.h>
       char *strchr(const char *s, int c);

DESCRIPTION
       The  strchr()  function  returns  a pointer to the first occurrence of the character
       string s.  Here "character" means "byte"; this function does not work with wide or m

RETURN VALUE
The strchr() function returns a pointer to the matched character or NULL if the character
```

Appendix II: ASCII

NAME

ascii − ASCII character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

ASCII is the American Standard Code for Information Interchange. It is a 7-bit code. Many 8-bit codes (such as ISO 8859-1, the Linux default character set) contain ASCII as their lower half. The international counterpart of ASCII is known as ISO 646.

The following table contains the 128 ASCII characters.

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL ’\0’	100	64	40	@
001	1	01	SOH (start of heading)	101	65	41	A
002	2	02	STX (start of text)	102	66	42	B
003	3	03	ETX (end of text)	103	67	43	C
004	4	04	EOT (end of transmission)	104	68	44	D
005	5	05	ENQ (enquiry)	105	69	45	E
006	6	06	ACK (acknowledge)	106	70	46	F
007	7	07	BEL ’\a’ (bell)	107	71	47	G
010	8	08	BS ’\b’ (backspace)	110	72	48	H
011	9	09	HT ’\t’ (horizontal tab)	111	73	49	I
012	10	0A	LF ’\n’ (new line)	112	74	4A	J
013	11	0B	VT ’\v’ (vertical tab)	113	75	4B	K
014	12	0C	FF ’\f’ (form feed)	114	76	4C	L
015	13	0D	CR ’\r’ (carriage ret)	115	77	4D	M
016	14	0E	SO (shift out)	116	78	4E	N
017	15	0F	SI (shift in)	117	79	4F	O
020	16	10	DLE (data link escape)	120	80	50	P
021	17	11	DC1 (device control 1)	121	81	51	Q
022	18	12	DC2 (device control 2)	122	82	52	R
023	19	13	DC3 (device control 3)	123	83	53	S
024	20	14	DC4 (device control 4)	124	84	54	T
025	21	15	NAK (negative ack.)	125	85	55	U
026	22	16	SYN (synchronous idle)	126	86	56	V
027	23	17	ETB (end of trans. blk)	127	87	57	W
030	24	18	CAN (cancel)	130	88	58	X
031	25	19	EM (end of medium)	131	89	59	Y
032	26	1A	SUB (substitute)	132	90	5A	Z
033	27	1B	ESC (escape)	133	91	5B	[
034	28	1C	FS (file separator)	134	92	5C	\ ’\\’
035	29	1D	GS (group separator)	135	93	5D	]
036	30	1E	RS (record separator)	136	94	5E	^
037	31	1F	US (unit separator)	137	95	5F	_
040	32	20	SPACE	140	96	60	`
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27		147	103	67	g
050	40	28	(	150	104	68	h
051	41	29	)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	−	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	[illegible]

```
1 Multiple choices
A: 4
B: 2
C: 1, 2, 3, 4
D: 2
E: 3

2. C Basics
a. 0x00040000

b.
char
Bin2Hex(unsigned char c)
{
   if (c >= 0 && c < 10) {
      return '0'+ c;
   } else if (c >=10 && c < 16) {
      return 'a' + (c-10);
   } else {
      //error
      assert(0);
   }
}

c.

char *
EncodeToHex((unsigned char *)buf, int size)
{
   char *result = (char *)malloc(2*size+1);
   for (int i = 0; i < size; i++) {
      result[2*i] = Bin2Hex(result[i]>>4); //convert the higher-order 4-bits
      result[2*i+1] = Bin2Hex(result[i]&0xf); //convert the lower-order 4-bits
   }
   result[2*size] = '\0';
}
```

3. C and Buffer Overflow

a.
```
int
ParseGrade(char *input, char *name, int *grade)
{
   char *semicolon = strchr(input, ';')
   if (!semicolon)
      return 1;
   int name_len = (int)(semicolon - input);
   for (int i = 0; i < name_len; i++ ){
      name[i] = input[i];
   }
   name[name_len] = '\0';
   *grade = atoi(semicolon + 1);
}
```

b. sub $0x8,%rsp
   0x00...0400651

c. add $0x8,%rsp
   0x00...04006b7

d. 0x00...04006b7
   return address for ParseOneLineExample

e. 0x7ff...ffe510 [= 0x7ff...ffe528 - 0x28 + 0x10]

f. From the answers to (d) and (e), we know that the beginning of the buffer "name" is stored at 24-byte away (lower) than the memory address storing the return address.

Thus, the input string needs to be at least 24 bytes long and the last 8-byte of the 24-byte input should contain the address 0x00...0400696 [address of MyGreatHack].

4. Assembly

a. a : 0x7ff...fe510
   b : 0x7ff...fe508

b.
```
void foo(int *a, int **b) {
   *b = a + 1;
}
```

c.
```
void bar(int *a) {
   (*a) += 1;
}
```

d. foo(a, &b); bar(b)

e. 3