CSE 351 Section 6 – Buffer Overflow and Caches

Hi there! Welcome back to section, we're happy that you're here ☺

Buffer Overflow!

Consider the following C program:

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

int read_input() {
  char buf[8];
  gets(buf);
  return 0;
}
```

Here is a diagram of the stack at the beginning of the call to read_input():

Address	Value (hex)
%rsp+15	00
%rsp+14	00
%rsp+13	00
%rsp+12	00
%rsp+11	00
%rsp+10	~~40~~ 00 (null terminator)
%rsp+9	~~AF~~ 73
%rsp+8	~~3B~~ 72
%rsp+7	71
%rsp+6	70
%rsp+5	6F
%rsp+4	6E
%rsp+3	6D
%rsp+2	6C
%rsp+1	6B
%rsp+0	6A

a) What is the value of the return address stored on the stack?

0x40AF3B

Assume that the user inputs the string "`jklmnopqrs`"

b) Write the values in the stack before the "return 0;" statement is executed. Cross out the values that were overwritten and write in their new values.
(Hint: use the ASCII table at the bottom to convert from letters to bytes)

c) What is the new return address after the call to gets()?

0x7372

d) Where will execution jump to after the "`return 0;`"?

It will try to jump to 0x7372, but it will crash with a segfault

e) How many characters would we have to enter into the command line to overwrite the return address to `0x6A6B6C6D6E6F`?

14 = 8 for padding (the length of buf) + 6 for the length of the address in bytes. A null terminator is appended, but it's okay because the upper bytes were going to be 0x00 anyway

f) Create a string that will overwrite the return address, setting it to `0x6A6B6C6D6E6F`

"`abababab onmlkj`" (The first 8 characters don't matter since they're just padding)

In Lab 3, we are given a tool called `sendstring`, which converts hex digits into the actual bytes

```
>echo "61 62 63" | ./sendstring
abc
```

g) If we want to overwrite the return address to a stack address like 0x7FFFFFAB1234, we need to use a tool like sendstring to send the correct bytes.
Why can't we just manually type the characters like we did earlier with "`jklmnopqrs`"?

There is no character in ASCII we can type that will give us a byte value of 0x7F, 0xFF, or 0x12

Char	Hex
a	61
b	62
c	63
d	64
e	65
f	66
g	67
h	68
i	69
j	6A
k	6B
l	6C
m	6D
n	6E
o	6F
p	70
q	71
r	72
s	73
t	74
u	75
v	76
w	77
x	78

Check out the Lab 3 video on Phase 0 before you start the lab!
It's linked on the Lab 3 page

Caches: Locality!

Recall that we have two types of locality that we can have in code:

Temporal locality: when recently referenced items are likely to be referenced again in the near future.
Spatial locality: when nearby addresses tend to be referenced close together in time.

For each type of locality, can you give an example of when we might see it in code?

Temporal Locality:

Accessing a sum counter over and over; reading and writing to the same variable; etc.

Spatial Locality:

Accessing a[0] in an array, then a[1], then a[2] in order; accessing the first field in a struct, then the second, then the third; etc.

Accessing a Cache (Hit or Miss?)

Assume the following caches all have block size $K = 4$ and are in the current state shown (you can ignore "—"). All values are shown in hex. Tag fields are padded, while bytes of the cache blocks are shown in full. The word size for the machine with these caches is 12 bits (i.e. addresses are 12 bits long)

Direct-Mapped:

Set	Valid	Tag (8 bits)	B0	B1	B2	B3
0	1	15	63	B4	C1	A4
1	0	—	—	—	—	—
2	0	—	—	—	—	—
3	1	0D	DE	AF	BA	DE
4	0	—	—	—	—	—
5	0	—	—	—	—	—
6	1	13	31	14	15	93
7	0	—	—	—	—	—

Set	Valid	Tag (8 bits)	B0	B1	B2	B3
8	0	—	—	—	—	—
9	1	00	01	12	23	34
A	1	01	98	89	CB	BC
B	0	1E	4B	33	10	54
C	0	—	—	—	—	—
D	1	11	C0	04	39	AA
E	0	—	—	—	—	—
F	1	0F	FF	6F	30	0

Offset bits: **2**

Index bits: **4**

Tag bits: **6**

	Hit or Miss?	Data returned
a) Read 1 byte at `0x7AC`	Miss	—
b) Read 1 byte at `0x024`	Hit	0x01
c) Read 1 byte at `0x99F`	Miss	—

2-way Set Associative:

Set	Valid	Tag (8 bits)	B0	B1	B2	B3
0	0	—	—	—	—	—
1	0	—	—	—	—	—
2	1	03	4F	D4	A1	3B
3	0	—	—	—	—	—
4	0	06	CA	FE	F0	0D
5	1	21	DE	AD	BE	EF
6	0	—	—	—	—	—
7	1	11	00	12	51	55

Set	Valid	Tag (8 bits)	B0	B1	B2	B3
0	0	—	—	—	—	—
1	1	2F	01	20	40	03
2	1	0E	99	09	87	56
3	0	—	—	—	—	—
4	0	—	—	—	—	—
5	0	—	—	—	—	—
6	1	37	22	B6	DB	AA
7	0	—	—	—	—	—

Offset bits: **2**

Index bits: **3**

Tag bits: **7**

	Hit or Miss?	Data returned
a) Read 1 byte at `0x435`	Hit	0xAD
b) Read 1 byte at `0x388`	Miss	—
c) Read 1 byte at `0x0D3`	Miss	—

Fully Associative:

Set	Valid	Tag (12 bits)	B0	B1	B2	B3
0	1	1F4	00	01	02	03
0	0	—	—	—	—	—
0	1	100	F4	4D	EE	11
0	1	077	12	23	34	45
0	0	—	—	—	—	—
0	1	101	DA	14	EE	22
0	0	—	—	—	—	—
0	1	016	90	32	AC	24

Set	Valid	Tag (12 bits)	B0	B1	B2	B3
0	0	—	—	—	—	—
0	1	0AB	02	30	44	67
0	1	034	FD	EC	BA	23
0	0	—	—	—	—	—
0	1	1C6	00	11	22	33
0	1	045	67	78	89	9A
0	1	001	70	00	44	A6
0	0	—	—	—	—	—

Offset bits: **2**

Index bits: **0**

Tag bits: **10**

	Hit or Miss?	Data returned
a) Read 1 byte at `0x1DD`	Hit	0x23
b) Read 1 byte at `0x719`	Hit	0x11
c) Read 1 byte at `0x2AA`	Miss	—

Cache Sim

If you need help on using the cache sim, take a look at additional supplemental material that will guide you through using the cache sim (posted with today's section handouts)! The cache sim is very useful for lab 4 and corresponding homework assignments.