Quiz week 14

Question 5 (0 / 4 pt; mean 0.16)

Consider the following C code:

```
struct Files all_files[NUM_FILES];
int GetLastByteOfFile(int index) {
    if (index >= 0 && index < all_files) {
        struct File *file = &all_files[index];
        if (file->type == MEMORY) {
            return file->data[file->size - 1];
        } else if (file->type == DISK) {
            return GetLastByteOfDiskFile(file)
        } else {
            return -1;
        }
    } else {
        return -1;
    }
}
```

If the above function runs in kernel mode, we might be able to use a Spectre-style attack where the cache evictions caused by memory access of `file->data[file.size - 1]` allows us learn about the value of an arbitrary memory location. To perform this attack, the attacker would prefer to choose an out-of-bounds `index` such that ___.

A.	88%		the address of `all_files[index].data[file.size - 1]` is the memory address whose value they want to learn about
B.	2%		the address of `all_files[index].type` is the memory address they want to learn about
C.	4%	T (correct)	the address of `all_files[index].size` is the memory address they want to learn about
D.			the value of `all_files[index].type` is `DISK`

- file = &all_files[index]
- So if we provide an out of bounds index, we can read arbitrary memory
- file->data[], ie all_files[index].data[] is like array2
- file->size, ie all_files[index].size, is like array1

```c
if (x < array1_size) {
    y = array2[array1[x]];
}
```

our template

```c
void SomeSystemCallHandler(int index) {
    if (index > some_table_size)
        return ERROR;
    int kind = table[index];
    switch (other_table[kind].foo) {
        ...
    }
}
```

actual code

- kind ~ file->size, ie all_files[index].size, is like array1
  - This is the address we want to learn about by observing its cache behavior
- other_table [] ~ file->data[], ie all_files[index].data[] is like array2
  - Not using the .foo here

Question 5 (0 / 4 pt; mean 0.16)

Consider the following C code:

```
struct Files all_files[NUM_FILES];
int GetLastByteOfFile(int index) {
    if (index >= 0 && index < all_files) {
        struct File *file = &all_files[index];
        if (file->type == MEMORY) {
            return file->data[file->size - 1];
        } else if (file->type == DISK) {
            return GetLastByteOfDiskFile(file)
        } else {
            return -1;
        }
    } else {
        return -1;
    }
}
```

If the above function runs in kernel mode, we might be able to use a Spectre-style attack where the cache evictions caused by memory access of `file->data[file.size - 1]` allows us learn about the value of an arbitrary memory location. To perform this attack, the attacker would prefer to choose an out-of-bounds `index` such that ___.

A.	88%	○	the address of `all_files[index].data[file.size - 1]` is the memory address whose value they want to learn about
B.	2%	○	the address of `all_files[index].type` is the memory address they want to learn about
C.	4%	T (correct) ○	the address of `all_files[index].size` is the memory address they want to learn about
D.		○	the value of `all_files[index].type` is `DISK`

- Why not A?
- all_files[index].data is like array2
  - Based on which cache set is affected by different index values, we learn what those index values are
  - So we need to set up the index (~array1) to refer to the memory location we're interested in – here file.size, ie all_files[index].size

Q4

Consider the following code:

```
unsigned char check_array[32768];
int mystery = /* unknown */;

int Check(int key) {
    return check_array[(key + mystery) % 32768];
}
```

Suppose that:

- (to simplify the problem) virtual memory is not use
- `check_array` is located at physical address `0x1400000`
- the system has a 2-way 64KB (2 to 16 byte) data cache with 64-byte cache blocks.
- `Check` is compiled to perform exactly three memory accesses:
    - to read the global variable read `mystery`
    - to reads its return address from the stack
    - to read from `check_array`

Suppose we determine that calling `Check` evicts from the data cache sets as follows:

```
key value | evicts values from cache set indexes
-----------------------------------------------
0         | 15, 72, 435
32        | 15, 72, 435
48        | 15, 72, 436
128       | 15, 72, 437
8240      | 15, 52, 72
```

Based on this information what is a possible value for `mystery`?

- (0 + mystery) // 64 +/- multiple of 512 (number of cache sets) = 435
- (32 + mystery) // 64 +/- multiple of 512 (number of cache sets) = 435
- (48 + mystery) // 64 +/- multiple of 512 (number of cache sets) = 436
- ...

There was a typo originally that we fixed. Originally wrote %16384 instead of %32768, and 4096 instead of 8240 for the last row

Suppose we determine that calling `Check` evicts from the data cache sets as follows:

```
key value | evicts values from cache set indexes
------------------------------------------------
0         | 15, 72, 435
32        | 15, 72, 435
48        | 15, 72, 436
128       | 15, 72, 437
8240      | 15, 52, 72
```

Based on this information what is a possible value for `mystery`?

Answer:

Key: `a value between 27856 and 27871 +/- any multiple of 32768`

originally we erroneously wrote % 16384 instead of % 32768, and 4096 instead of 8240 for the last value

(0 + mystery) // 64 +/- multiple of 512 (number of cache sets) = 435

(32 + mystery) // 64 +/- multiple of 512 (number of cache sets) = 435

(48 + mystery) // 64 +/- multiple of 512 (number of cache sets) = 436

...

let's choose the mulitple of 512 to be 0 for simplicity, for now

mystery // 64 = 435 implies mystery in [435 * 64 = 27840, 27840 + 63 = 27903]

(32 + mystery) // 64 = 435 implies mystery in [435 * 64 - 32 = 27808, 27808 + 63 = 27871]

(48 + mystery) // 64 = 436 implies mystery in [436 * 64 - 48 = 27856, 27856 + 63 = 27919]

overlap here implies mystery in [27856, 27871]

the multiple of 512 offsets this by 512 * 64 = 32768