

## McDonnell – Analyzing Simple Buffer Overflow Conditions

### Code Problem

First to understand what's happening here we must analyze our memory structure as follows.

We first declare a 10 element 4-byte integer buffer whose address structure and stack space of is highlighted in green here:

0x00007fffffffe240	64 00 00 00	65 00 00 00	66 00 00 00	67 00 00 00
0x00007fffffffe250	68 00 00 00	69 00 00 00	6a 00 00 00	6b 00 00 00
0x00007fffffffe260	6c 00 00 00	6d 00 00 00	6e 00 00 00	01 00 00 00

Next, the green arrow is pointing to the single 4-byte address space for our incremental integer j:

0x00007fffffffe240	64 00 00 00	65 00 00 00	66 00 00 00	67 00 00 00
0x00007fffffffe250	68 00 00 00	69 00 00 00	6a 00 00 00	6b 00 00 00
0x00007fffffffe260	6c 00 00 00	6d 00 00 00	6e 00 00 00	01 00 00 00

This is easily verified in our memory viewer by selecting the address of j which gets highlighted below:

Go to:	&j	▼	View	0x00007fffffffe268	
0x00007fffffffe180	00 00 00 00	00 00 00 00	01 00 00 00	00 00 00 00	.....
0x00007fffffffe190	65 00 00 00	00 00 00 00	01 00 00 00	00 00 00 00	e.....
0x00007fffffffe1a0	44 e2 ff ff	ff 7f 00 00	0c e1 ff ff	ff 7f 00 00	D.....
0x00007fffffffe1b0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....
0x00007fffffffe1c0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....
0x00007fffffffe1d0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....
0x00007fffffffe1e0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....
0x00007fffffffe1f0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....
0x00007fffffffe200	40 00 00 00	00 00 00 00	00 00 10 00	00 00 00 00	@.....
0x00007fffffffe210	10 00 00 00	00 00 00 00	40 00 00 00	00 00 00 00	.....@.....
0x00007fffffffe220	00 00 00 00	00 00 00 00	20 e2 ff f7	ff 7f 00 00	.....
0x00007fffffffe230	01 00 00 00	00 00 00 00	9d 11 40 00	00 00 00 00	.....@.....
0x00007fffffffe240	64 00 00 00	65 00 00 00	66 00 00 00	67 00 00 00	d...e...f...g...
0x00007fffffffe250	68 00 00 00	69 00 00 00	6a 00 00 00	6b 00 00 00	h...i...j...k...
0x00007fffffffe260	6c 00 00 00	6d 00 00 00	6e 00 00 00	01 00 00 00	l...m...n.....
0x00007fffffffe270	01 00 00 00	00 00 00 00	b0 fe c3 f7	ff 7f 00 00	.....
0x00007fffffffe280	00 00 00 00	00 00 00 00	36 11 40 00	00 00 00 00	.....6.@.....
0x00007fffffffe290	00 00 00 00	01 00 00 00	88 e3 ff ff	ff 7f 00 00	.....
0x00007fffffffe2a0	00 00 00 00	00 00 00 00	03 6d 7b 38	97 14 73 9c	.....m{8...s...
0x00007fffffffe2b0	88 e3 ff ff	ff 7f 00 00	36 11 40 00	00 00 00 00	.....6.@.....
0x00007fffffffe2c0	08 3e 40 00	00 00 00 00	00 d0 ff f7	ff 7f 00 00	>@.....
0x00007fffffffe2d0	03 6d 79 fd	68 eb 8c 63	03 6d b1 c4	10 fb 8c 63	...my...h...c...m...c...
0x00007fffffffe2e0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	.....
0x00007fffffffe2f0	00 00 00 00	00 00 00 00	01 00 00 00	00 00 00 00	.....
0x00007fffffffe300	88 e3 ff ff	ff 7f 00 00	00 f6 59 0c	e6 3d 77 ae	.....Y...=w...

Next we verify the address of our single 4-byte address space for our incremental integer i:

Go to:	&i	View	0x00007fffffffe26c
0x00007fffffffe180	00 00 00 00 00 00 00 00	01 00 00 00 00 00 00 00	.....
0x00007fffffffe190	65 00 00 00 00 00 00 00	01 00 00 00 00 00 00 00	e.....
0x00007fffffffe1a0	44 e2 ff ff ff 7f 00 00	0c e1 ff ff ff 7f 00 00	D.....
0x00007fffffffe1b0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
0x00007fffffffe1c0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
0x00007fffffffe1d0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
0x00007fffffffe1e0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
0x00007fffffffe1f0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
0x00007fffffffe200	40 00 00 00 00 00 00 00	00 00 10 00 00 00 00 00	@.....
0x00007fffffffe210	10 00 00 00 00 00 00 00	40 00 00 00 00 00 00 00	.....@.....
0x00007fffffffe220	00 00 00 00 00 00 00 00	20 e2 ff f7 ff 7f 00 00	.....
0x00007fffffffe230	01 00 00 00 00 00 00 00	9d 11 40 00 00 00 00 00	.....@.....
0x00007fffffffe240	64 00 00 00 65 00 00 00	66 00 00 00 67 00 00 00	d...e...f...g...
0x00007fffffffe250	68 00 00 00 69 00 00 00	6a 00 00 00 6b 00 00 00	h...i...j...k...
0x00007fffffffe260	6c 00 00 00 6d 00 00 00	6e 00 00 00 01 00 00 00	l...m...n..._...
0x00007fffffffe270	01 00 00 00 00 00 00 00	b0 fe c3 f7 ff 7f 00 00	.....
0x00007fffffffe280	00 00 00 00 00 00 00 00	36 11 40 00 00 00 00 00	.....6...@.....

Which we can see is also sequentially assigned next to j and our integer buffer

0x00007fffffffe240	64 00 00 00	65 00 00 00	66 00 00 00	67 00 00 00
0x00007fffffffe250	68 00 00 00	69 00 00 00	6a 00 00 00	6b 00 00 00
0x00007fffffffe260	6c 00 00 00	6d 00 00 00	6e 00 00 00	01 00 00 00

Thus our problem begins with our over bounded loop:


```

15
16      /* initialize elements of array */
17      for (i=0; i<=10; i++){
18          n[i] = i + 100;  n: int [10]
19      }

```

Where the count is actually going beyond the 10 element array since counting zero + 10 is 11 elements we begin to write into the next address space or overflow into j as seen here:

```

>  n = {int [10]}
  10 01 i = {int} 11
  10 01 j = {int} 110

```

As seen above, j becomes the next element in the array unintentionally. Thus it's fair to say any calls beyond the array n[9] are actually calling the same memory address as int j and anything further eventually would call int i.

Since j and n[10] are the same memory space we get a bug where our later for loop whose incrementer is j begins to change values of n[10].

Here  
during  
the loop  
we  
display  
n[10] as

```
27      /* initialize elements of array */
28      for(j=0; j <= 10; j++){
29          printf(format: "Element[%d] = %d\n", j, n[j]);
30      }
31
32
```

the value being looped through by j which is 10 at that point in the loop state. Eventually, j increments to the total value of 11 (again  $0 + 10$ ) is 11 counted elements – thus once it finally exits the for loop the address space at j and n[10] is set to 11 which is why calling it after as seen here:

```
printf(format: "Element[%d] = %d\n", j, n[10]);
```

Is displaying out loops final count instead of its looping iteration value.

## Solution

The solution for this particular issue is kind of moot as it was a demonstration of bounds overflowing but the main issue is actually common and that is having an incorrect loop stop value that exceeds the buffer being looped through. It's important to remember to stop your loop at the appropriate point and not equal or exceed to a point beyond its declaration. So in this case changing :

```
/* initialize elements of array */  
for (i=0; i<=10; i++){  
    n[i] = i + 100;    n: int [10]  
}
```

To this:

```
16      /* initialize elements of array */  
17      for (i=0; i<10; i++){  
18          n[i] = i + 100;    n: int [10]  
19      }
```

Would fix the loop so it wont overwrite into j. The same for the display loop. Calling the array directly beyond its bounds would likely be intentional or a typo and the only fix would be not doing it.