Baggy Bounds (Contd.)
- Aligned and padded to $2^k$ bytes.

\[
p' = p + i; \text{ to check whether } p' \text{ points to same as } i.
\]

\[
\text{assert } ((p \text{ and } p') \gg BNDS [p\gg4]==0); \text{ Here } p \text{ and } p' \text{ points to same object.}
\]

This is arbitrary [points allocated to smaller is 16 bytes that is minimum allocation]

We do \( p \) and \( p' \) to see how many bits they differ which must be the lower half \( k \) bits of the allocation as depicted below. Here, upper 32-k bits points to the even size chunks of \( 2^k \) in the virtual address space, it specifies which of the chunks it belongs to, lower \( k \) bits will specify the offset within that chunk.

OUT OF BOUND POINTERS (OOB)

Concept using figure:-
Now let us see the snippet of code,

\[
P = q + i;
\]

if (!inbounds (p, q))
{
    p = p | 0x80000000;
    r = p - i;
    *r;
}

Baggy Bound Approach for snippet of code above would be:
p = q + i;

if(!inbounds (p.q))
    (but here p is within 2^k-1 of being in bounds)
1) p = p - 2^k (if p is too large) This help us resolve the object
   or
   p = p + 2^k (if p is too small)

2) Set MSB of p. This is done to point to the out of bound region or bring it back below.

CLEVER TECHNIQUE:-

Note:- If p do not point to this memory, out of bound aborts the prog.
FAT POINTERS:-

Idea: - Pointers carry bound information.

Example:

LANGUAGE BASED APPROACH:-

- For each pointer variable add a bounds variable.
- On pointer assignment assign corresponding bounds variable
- On pointer dereferencing check the bounds.

Let us see how it works with snippet of code:-

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Code Added by Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>void pcopy(int *a, int *b, int n)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>int *p=a;</td>
<td></td>
</tr>
<tr>
<td>for (i=0;i&lt;n; i++)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>*p++=*b++</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
| typedef struct bounds{
|   void *lo;
|   void *hi;
| } bounds;
| void(int *a, bounds a_b, int *b, bounds b_b, int n) |
| { |
|   int *p=a; |
|   bounds p_b=a_b;
|   for (i=0;i<n; i++) |
|   { |
|     assert(p_b.lo <= p < p_b.hi);
|     assert(b_b.lo <= b < b_b.hi);
|     *p++=*b++ |
| } |

Here, Prototype Compatibility issue creeps in as function name is same with difference in arguments

Considering another snippet of code for list:-
Note:- In this case, Size of the structure keeps on increasing or changing as we keep introducing the bounds, so it is a problem. To overcome this, we create a shallow struct.

```c
struct list_shallow{
    bounds b_next; // Bounds b;
    or
    struct list_shadow *next;
};
```

Therefore now, program changes

```c
void traverse (struct *l, struct list_shadow b_l)
{
    while(l){
        boundscheck(&l->data, b_l.b);
        b_l=*b_l.next;
        l=l->next;
    }
}
```

So, the idea is if the original program has a link list of data, new program will have the same link of data, but it will also have link list of bounds.
So, there will be a simultaneously traversal of the bound link list and original data link list. So, whenever we insert link list for data, we insert corresponding link list for bound. So if we have original list of tree then we will have tree for bound variables too.

Data

Exactly mirror shape

Bounds

So, if we have L pointing to the link list as depicted below, l_b will point to another link list for bound.

Why do we not need bounds for the bound structure that we use?
Because, this is inserted by the compiler, so we don’t need struct for these. We assume that people who wrote compiler didn’t make any mistake.

Why do we need to store the bounds in list?
Suppose programmer make a silly mistake and does this in the example below:-
Void *p = malloc(1); // That's the silly thing he does. he allocates single byte of memory
b_p=(p, p+1); // they initialize like this
l->next = (struct list *)p; // programmer can surely do this
l_b.next=b=p_b; // bounds on l_b.next , we have copy p bounds on l->next
l=l->next;
l_b= b l.next

So, now if we dereference more than 1 bytes here, it will throw the error.

Class Concluded.