Store Bounds
    / \  
   by /  \  
pointer / \  \  By Region
    / \  
Soft Cured Approach
Bounds
(Hash table)

Last class, the discussion was about using hash tables to store bounds.

Misc Note:
Often, we have lots of libraries in an application. But only few of them are security critical. By transforming only those libraries, we avoid unnecessary runtime overhead.

**Cured Approach**

In the Cured approach, for every pointer, we maintain a shadow copy containing the bounds for that pointer.

Consider the following structure

```c
struct list {
    struct list *next;
    bounds b_next;
    int x;
}
```

Instead of storing the bounds as part of the list, we maintain a shadow for the struct list as follows

```c
struct list_shadow {
    bounds b_next;
    struct list_shadow *next;
}
```

Whenever a pointer is assigned, its bounds pointed to by the shadow copy are also updated.
Consider two pointers p and q, and let their bounds be b_p and b_q.
Their shadow copies are denoted by p_s and q_s.
So when an assignment p = q is done, we need to do the following
b_p = b_q
p = q
p_s = q_s
Whenever a pointer is dereferenced, it is checked against the bounds pointed by its shadow copy. When an operation such as \texttt{*p=0}, we need to do the following:

\begin{verbatim}
  b_check(b_p, p, p+1) – We check if p is within the bounds pointed to by b_p.
  \texttt{*p = 0;}
\end{verbatim}

If the pointers are part of list representation, then we also maintain a shadow list which contains the bounds for every pointer.

If we have a function which constructs and returns an object of type struct as follows:

\begin{verbatim}
struct list* cons(struct list *l, int x)
{
    struct list *t = malloc(sizeof(struct));
    t->x = x;
    t->next = l;
    return t;
}
\end{verbatim}

This code needs to be transformed to include the code that does the assignment of bounds to the new struct and also to check the dereferencing of the struct pointer. The transformed code is as follows:

\begin{verbatim}
struct list *, struct list_shadow* cons(struct list *l,
                                      bounds b_l,
                                      struct list_shadow *l_s,
                                      int x)
{
    struct list *t = malloc(sizeof(struct list));
\end{verbatim}
struct list_shadow t_s = malloc(sizeof(struct list_shadow)); - Allocate a shadow copy

bounds b_t = {t, t+1}; - Initializing the bounds for the struct list t;
bcheck(b_t, t, t+1); - Check before pointer dereferencing of 't'
t->x = x;

bcheck(b_t, t, t+1);
t_s->b_next = b_l; - Update bounds of t_s to bounds of l
t->next = l;
t_s->next = l_sl - Update the shadow list pointer to point to shadow copy of 'l'.

return t, t_s; - Return the structure 't' and its shadow copy 't_s'

How is the shadow copy protected from corruption?
Since every pointer is checked against its bounds before dereferencing, it is not possible that any program trying to access memory will overwrite the shadow copy.

What happens if a buffer overflow occurs?
In Ccured, if a buffer overflow occurs, it results in a DoS. But this is better than allowing unauthorized access to memory.

What happens if we work with an untransformed library?
Consider the struct list example

- If the library does not modify the pointer but only dereferences it, then its ok.
- If the library introduces a node in the list, then this approach will not work.

Jones & Kelly Approach

- pointer bounds are associated with region pointer points to

Consider the following allocation
int *p = malloc(n);

The bounds for 'p' is stored as follows
bounds_store(p, p+n);

The bounds for 'p' is retrieved by using a look up function as follows
bounds_lookup(p);

When 'p' is dereferenced, the following is done to check the bounds
bounds t = bounds_lookup(p);
bcheck(t, p, p+1);
*p =0;
Techniques for bounds_Store

- Use a hash table

<table>
<thead>
<tr>
<th>p</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>n</td>
</tr>
</tbody>
</table>

This will not work if we do p++ and then do *(p++).
So, we can store an entry for every 't' belongs to the range p, p+n. This means too much memory wastage.
So, hash tables – Bad idea.

- Interval Tree

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p</td>
</tr>
</tbody>
</table>

Provides Lookup and updates in O(logn)

When we do p++ and then do bounds_lookup(p) we will end up checking the bounds for a diff region.

To overcome this, we can do the following,

bounds t = bounds_lookup(p);
p++;
bcheck(t, p, p); // check if address from p to p is within the bounds t. So we are actually checking if p++ is within the region bounds of 'p'.

Consider the following code
p = p+(q-r);
If compiler optimizes the above and changes the code to
p = p+q;
p = p-r;

Then, the bounds check for the statement p = p+q, will fail the program when p is incremented by q(if 'q' is large enough to make 'p' point to an out of bounds memory).