
Baggy bounds checking:

- Metadata
  - Extra information that is kept by the program related to other data.

Example of possible unsafe code:

```c
void foo(char* p)
{
    p[5] = 0;
}
```

- C does not keep track of data to provide memory safety.

**Challenge:** Track bonds of pointers to prevent attacks.

<table>
<thead>
<tr>
<th>Pointer operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>dereferences</td>
<td>*p</td>
</tr>
<tr>
<td>arithmetic</td>
<td>q = p + s</td>
</tr>
<tr>
<td>allocation</td>
<td>heap - p = malloc()</td>
</tr>
<tr>
<td></td>
<td>stack - p = &amp;x</td>
</tr>
<tr>
<td>deallocation</td>
<td>free(p)</td>
</tr>
<tr>
<td>pass to functions</td>
<td>foo(p)</td>
</tr>
<tr>
<td>function pointers</td>
<td>*function()</td>
</tr>
<tr>
<td>casting</td>
<td>char <em>p (int</em>)p</td>
</tr>
<tr>
<td>assignment</td>
<td>p = q</td>
</tr>
<tr>
<td>literal</td>
<td>NULL, 1234</td>
</tr>
<tr>
<td>pointer to int cast</td>
<td>int x = (int)p</td>
</tr>
</tbody>
</table>
Storing bounds with the pointer:

Example:

- Increase the word size of a 32bit computer to 96bits.
- Store upper and lower bounds in newly created 64bit area.
- The address space is still 32bit.
- Primitive data types have \{0, 0\} as their bounds.

**We want to verify that:**

\[ \text{lo} \leq p + \text{sizeof}(*p) \leq \text{hi} \] ← This is the test to determine if we are within the bounds.

<table>
<thead>
<tr>
<th>32bits</th>
<th>32bits</th>
<th>32bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>lo</td>
<td>hi</td>
</tr>
<tr>
<td>int</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>96bit word</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Problems that occur with pointers:

1. Spatial violation – Pointer that goes out of bounds.
2. Temporal violation – Pointer that points to an unallocated address space.

**Spatial Violation Example**

```c
p = malloc (10);
p +=11;
*p;
```

**Temporal Violation Example**

```c
p = malloc (10);
qu = p;
free(p);
*q = 0;
```

Soft bounds-

Bound information is stored in a hash table indexed by the address of the pointer (&p)

```
... ... ...
&p lo hi ...
... ... ...
```
Example:

```c
int a[10];  // Make an array of size 10
htable_insert(&a,a,&a[10]);  // Insert the bounds information into the hash table
int *p = a;  // Assign a pointer to the array
bound b = htable_lookup(&a);  // Obtain the bounds information about a from the hash table
htable_insert(&p,b_lo,b_hi);  // Insert the bounds information about p into the hash table
```

Contents of the hash table after operations above:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>&amp;a[10]</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;p</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example continued:

```c
b = htable_lookup(&p);  // Obtains the bounds from the table
assert(b_lo <= p && p + sizeof(p) <= b_hi)  // Check if p is within those bounds, if not stop the execution
htable_insert(&p,0,0)  // Insert new information about p bounds
p = (…) 12;  // Perform an operation on p
```
Using a pointer to hold bounds information

```c
int a[10]  // Make a new array of size 10
bounds b_p = {&a, &a[10]};  // Set the bounds for that array
int *p = a;  // Make a new pointer that points to the array

int *q;  // Make a new pointer
bounds b_p;  // Make a new set of bounds

assert(b_p.lo <= p && p + sizeof(p) <= b_p.hi);  // Check that p is within the bounds
*p = 0;  // Set what p points to as 0
b_p = {0, 0};  // Set the bounds for p to 0, 0
p = (…) 12;  // Operation on p
b_q = b_p;  // Set the bounds of q equal to the bounds of p
q = p;  // Set q equal to p
```
Using a heap to hold bounds information

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

Where do you store the bounds? In the list?

struct * list cons(struct* list l, int x, bounds b_l)
{
    struct list* t = malloc (......);  // Allocate space for the new node t
    bound b_t = {t, t + 1};          // Set the bounds for t
    t -> b_next = {0,0};            // Set the bounds for the next node as 0,0
    assert ( .....bounds check for t...); // Check the bounds of t
    t -> x = x;                     // Set the value inside the node
    assert ( .....bounds check for t...); // Check the bounds of t
    t -> next = l;                  // Set the next in t to point to the existing list
    t -> b_next = b_l;              // Set the bounds for the next to the bounds of the existing list l
    return t;                      // return the list
}