Bounds – Checking C [Jones and Kelly]

Every time memory is allocated, the beginning and end of the variable (the range of the object's address) are stored in a data structure.

- For efficient access, a Splay Tree is used
- Pointer arithmetic and dereference operations involve a search, so speed is important to minimize overhead

This is done for each static variable at the beginning of the program.
At the entry of each function, memory regions corresponding to each of its local variables need to be inserted into splay tree; just before return, these regions have to be deleted from the tree. Finally, the region returned by each malloc operation needs to be entered into the tree, and the free operation should delete that region from the tree.

Whenever a pointer is dereferenced, the pointer value is checked in the splay tree to see if it corresponds to a valid object. If not a memory error is flagged and the program aborted.

Just checking pointer dereferences is not enough: it wont capture out-of-bounds access, since the access would still correspond to a valid object, but not the intended object. (Typically, an out-of-bounds array access ends up accessing an adjacent object in memory.) At the point of dereferencing, there is simply no information to help determine if the target being accessed is the intended object: we have just a single pointer value, which has already advanced beyond the intended object. So, the only way to check if a pointer is going out-of-bounds is to "catch it in action." In particular, pointer arithmetic operations need to be checked to determine if they cross an object boundary, and if so, an error should be flagged.

Consider a pointer-arithmetic operation: q + e, where q is a pointer variable and e is an integer expression. To prevent out-of-bounds access, we need to check that q as well q+e fall within the memory region corresponding to a single object. Otherwise an error is flagged.

The problem with this approach is that it performs "eager checking" of pointer values. In particular, it seems reasonable to perform arbitrary pointer arithmetic if the result is never dereferenced. So, a better alternative is to simply set the pointer value to an invalid value rather than immediately flagging an error when the pointer arithmetic leads to an out-of-bounds pointer. Unfortunately, even this does not always work with all programs.

Consider the following loop that initializes all elements of an array with zero, except the last element which is set to 1.

```
int a[n];
for (p = a; p < &a[n]; p++)
        *p = 0;
p--; *p = 1;
```

Note that when the loop is exited, p goes outside the range of array a. Thus, the last pointer arithmetic operation p++ will lead to an out-of-bounds pointer. If p is reset to an invalid value (e.g., NULL), subsequent operation *p = 1 will lead to a runtime error. However, there is no memory error in this program.

To avoid raising errors in correct programs, Bounds-Checking C extends arrays by 1 element. Unfortunately, this can still lead to problems in the following variant of the same loop:

```
int a[n];
for (p = a; p < &a[n]; p += 4)
        *p = 0;
p -= 4; *p = 1;
```

In this case, every 4^{th} element is initialized to zero. In this case, the last pointer arithmetic may take the pointer value to as many as 3 elements beyond the end of the array a. Since the padding is only one element, Bounds-checking C will result in a runtime error on the above program.

CRED [Ruwase et al]

To avoid the problem experienced by bounds-checking C, a simple technique was proposed by Ruwase et al and was implemented into a system called CRED. If a pointer goes out-of-bounds, it is set to point to a special out-of-bounds object that is stored elsewhere in memory, for instance in high addresses. These special out-of-bounds objects are designed to store information about the original pointer, most importantly the base address, the bound, and the value of the pointer when it was changed to the out-of-bounds object. The purpose of these special objects is to store information about a pointer so that it can be later restored, for instance when it returns back within the bounds of the original object.

For instance, an object starts at address 100 and ends at 200. A pointer p points to this object. The program temporarily assigns p to 204, meaning that it cannot be de-referenced (since it would be out-of-bounds of the object). So, a metadata object is created to store information about this pointer --- the metadata includes the base, bound and the current value of the pointer.
This metadata object is allocated in high memory --- for instance, on Linux, the maximum usable memory address is typically 3GB. The compiler can allocate a region next to the highest addressable memory to store these metdata objects, say, from 3GB downto 3GB-100MB. Each time a pointer goes out of bounds, a metadata object can be allocated in this memory region, and then the out-of-bounds pointer value is changed to point to its metadata.

The purpose of using a higher value for out-of-bounds pointer than any valid pointer is so that such out-of-bounds pointers can be easily recognized and processed in a special way. This special processing can be understood best in terms of a source-to-source transformation. Consider the following code:

```
p = p – 4;
*p = 5;
```

This gets transformed as follows:

```
if (p > max_valid_address) /* say, 3GB-100MB in the above example */ {
   p->value = p->value – 4;
   if ((p->value >= p->base) && (p->value <= p->bound)) {
      tp = p;
      p = p->value;
      free_oob_object(tp);
   }
}
else if (goes_out_of_bounds(p, p-4))
   p = allocate_new_oob_object(lookup_base(p), lookup_bound(p), p-4);

if (is_valid(p)) /* look up splay tree to determine if p's value falls within
                    the range of some valid object*/
       *p = 5;
else .... /* flag a runtime error */
```

Benefits and Drawbacks of Bounds-Checking approaches

The bounds-checking approaches described above are highly backward-compatible: if malloc/free calls are wrapped so that they insert/delete from the splay tree, then instrumented programs can work with uninstrumented libraries in most cases. In particular, in most such scenarios, the executable , which we are assuming to be instrumented, will end up calling the library. If the library returns an object, that will have been allocated typically on the heap. Since we wrapped malloc/free, information about such objects would still be in the splay tree. This is unlike other memory error detection techniques that typically require all libraries to be instrumented.
Note that the above benefit does not work in cases where an uninstrumented library passes a statically or stack-allocated object to instrumented code. (This is unusual.)

The drawback of bounds-checking approaches are as follows:
(a) pointer to integer casts may not work as expected. In particular, such casts may result in memory errors going undetected.
(b) pointer-to-pointer casts may implicitly result in pointer-to-integer or pointer-to-nonpointer casts if the the pointer objects are structs that in turn contain a combination of pointer and non-pointer fields.
(c) unions of pointer and non-pointer types lead to the same problem as (a), while unions consisting of different structure types lead to same problems as in (b)
(d) object-to-object copies may lead to a similar problem if the source and destination objects have different type, and contain a combination of pointer and nonpointer values.
(e) temporal errors that occur due to reallocated memory are not detected.

Example of case (b):

```
struct A{
        int i;
} a;
struct B {
        char *p;
} *b;
b = (struct B*)&a;
```

This code is permitted in C language, but causes the integer field value i to be interpreted as a pointer field value p. If p is subsequently dereferenced,

the best we can do is to check if p corresponds to a validly allocated object; we have no way to check if p was fabricated, or if it resulted from invalid pointer arithmetic operations.

Detecting all memory errors at runtime

Smart pointers – store base and bound values

- stored alongside data
- problem: messes up the layout of data structures
- as a result cannot link code with smart and regular pointers due to inconsistency

Compatibility is a major issue. E.g. a pointer with “in-band” metadata will be 20 bytes, while ordinary pointer has only 4 bytes. When you try to combine two pieces of codes with one of them having metadata and the other having not, problems arise: in particular, a structure containing pointers cannot be passed by instrumented code to an uninstrumented library since the two will have different sizes.

An alternative approach is to store metadata “out-of-band” --- that is, we store them separately. For instance, for each pointer variable p, we can create a variable p_info to store its metadata.

See slides from the lecture. Also see the paper at
http://seclab.cs.sunysb.edu/seclab/pubs/papers/fse04.pdf
You can ignore some of the more involved aspects, including casts and pointer arithmetic.