Lecture- 13 System Security

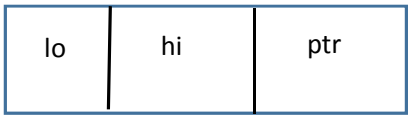
Lecturer: Rob Johnson

Date: 14 October 2013

Scribe: Chirag Mandot

⊙ Topics Discussed: **SoftBound, SoK: Eternal War in Memory**

- **SoftBound[** SoftBound: Highly Compatible and Complete Spatial Memory Safety for C, Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, Steve Zdancewic**]:** A compile time transformation for enforcing spatial safety of C. It associates base and bound metadata with every pointer and record the metadata in disjoint metadata space that is accessed via explicit table lookups, thus provides the memory layout compatibility.
- SoftBound Implementation can be done in the following ways:
  1. Local Variables: In this we add local bounds variables
  2. Heap Data:
     - Shadow Structure with bounds information
       or
     - Hash Table
- Fat Pointers can be represented as :



Fat pointers do create memory overhead, so why not store this information in hash table

So we can have bound table represented as

BNDS I &ptr → (lo,hi)

- The example will depict how it is used:

```
Struct list {
      Struct list * next;
      int data;
};
void traverse (Struct list *L, bounds l_b){
      while(L){
            bcheck(L,l_b) // it will check the bounds
            f(L→data);
            L=blookup(&L→next) // it will lookup in the hash table
            L=L→next;
      }
}
```

The code in red is added during compile time

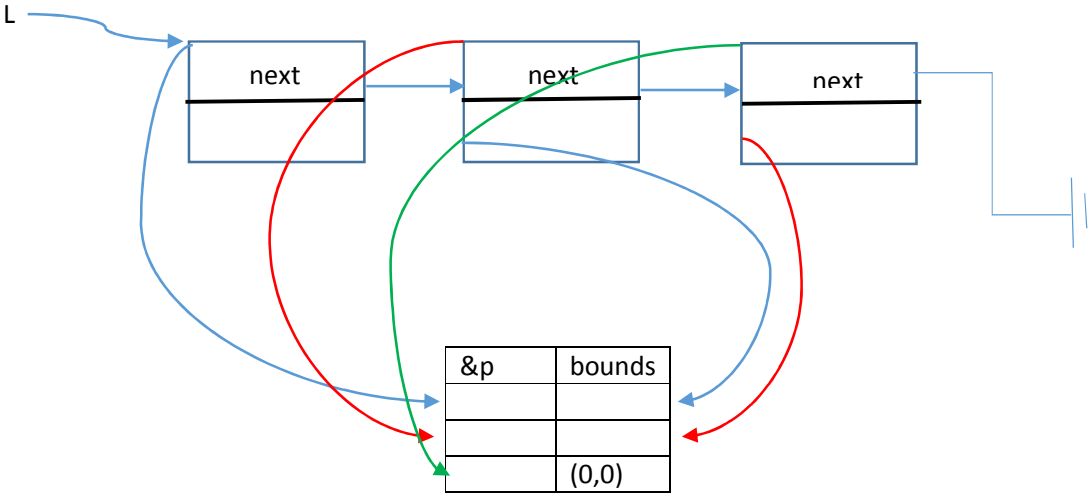
In the above code we saw a simple program and its compiled code.

- Now lets Prepend a list node(now node with a as element)

```
Struct list *prepad (Struct list * l, bounds l_b, int a){
      Struct list *tmp=malloc(….);
      bounds tmp_b=[tmp,tmp+1]; //We just increment to create space for the bounds
      blookup(&tmp→next);
      *tmp→next;//It will give error as heap is not initialized yet.Just memory is
allocated

      bcheck (tmp,tmp_b);// It will check the bounds of tmp variable
      tmp→data=a;
      BNDS[&tmp→next]=l_b;
      tmp→next=L;// points to pointer on heap but tmp is maintained on stack
      }
      Code in green: an error which a programmer might make
      Code in black : The actual code
      Code in red: Added by the compiler
```

- Following the above code we can draw a hash table for the list L. The table bellows shows the index and bounds



Size of next = l_b

- We can assign null to the last node in two ways:

1. Here as we have assigned zero in the hash table
2. Point it to null pointer

Q: Why not keep track of tmp at malloc itself?

1. Because it will lead to performance degradation
2. Heap table collision
3. Even an efficient table will lead to decrease in cache locality

❖ Reclamation of Hash Table:

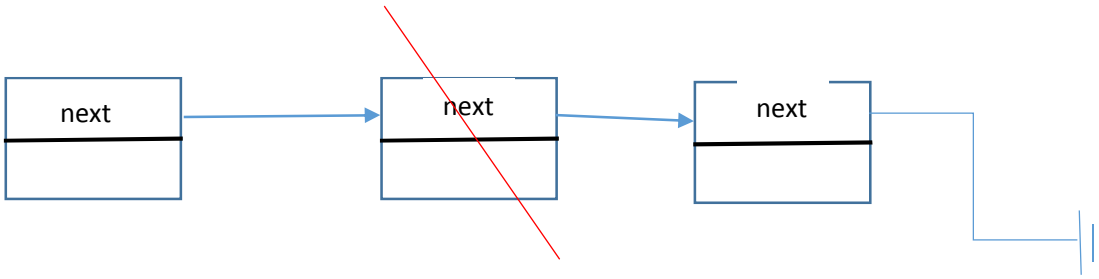
When free is called following is the output

Basically it should call the pointer p and just free it, but as we have list it will affect the neighboring to which it is pointing to.

Let the code be:

```
BNDS[&L→next→next]={0,0}
Free(L→next); // it will check the type and invalidate their pointer also. As it's a list, it has pointer
BNDS[&L→next]={0,0};
```

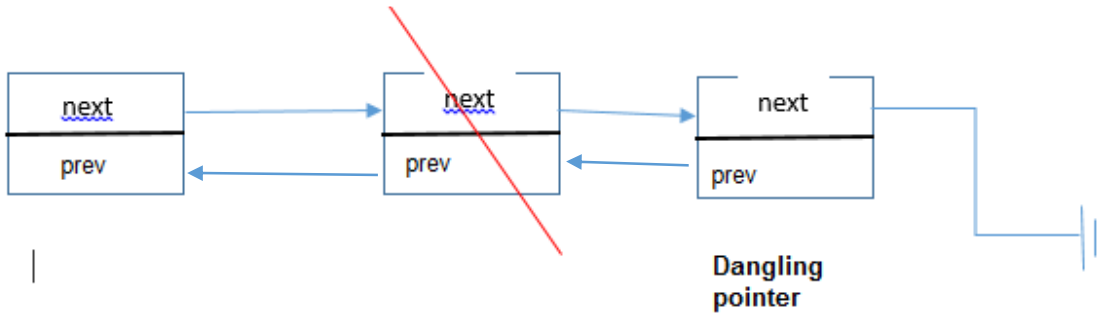
In the red is the code added later



❖ Procedure:

Here the when we make the second next null, the first next becomes dangling as the data it was pointing is deleted. Therefore, the compiler will automatically insert code to make first next to point to the third next which is already there in the bound table

In case of doubly linked list:



Here the middle next gets destroyed and the last node becomes dangling.

The code inserted will be as follows:

```
Bnds[&L-->next-->next]={0,0}
Bnds[&L-->next-->prev]={0,0}
free(L-->next);
Bnds[&L-->next]={0,0}
```

Note: In C we cant find all the pointers and destroy them. So in this method we look only for spatial errors and not temporal errors

❖ Evaluation Criterion:

1. **Effectiveness:**
   Very good as far as spatial safety is concerned
2. **Performance:**
   It adds 67% overhead and 22 % in store only checking mode
3. **Memory overhead:**
   Doesn't matter much
4. **False positive:**
   Very good. As if we point to the end of an array also then its ok as we don't reference it. It can also happen during narrowing the pointer bounds on user request. Example of false positives could be seen in Apache test. Note: Dangling pointer is false negative
5. **Separate compilation:**
   All lines of codes are independent and self-defined. We just insert before and after, so it's supported
6. **Code changes:**
   Very less code changes and hence good.
7. **Linking**
   Moderate. As when we compile we create parallel library function with different parameter
   Prepend(lib fn)→sb_prepend(compiler added fn)
   So if the program while using softbound calls our function then there is no problem, but if calls library function then softbound wont take place. Hence Moderate

8. **Compatibility:**
   The compatibility is not good as the inserted code snippets are not universal. So difficult implementation

❖ SoK: Eternal War in Memory:

Defense mechanism discussed in class:

1. **Address Space Layout Randomization(ASLR):**
   ASLR reduces the overhead and its very compatible as OS can do it instead of small softwares
2. **NX Bits:**
3. It cant be done at application level. OS can do it easily. Reduces overhead and effective
4. **Stack Canary:**
   Compiler do the changes