Buffer overflow defenses

There are two categories of buffer-overflow defenses:
- “Make it hard for the attacker to exploit buffer overflow”
  - Address space layout randomization
  - Model checking to catch abnormal execution
- “Maintain bounds information for each buffer or pointer in the program so that overflows are caught early”
  - Fat pointers
  - Deputy

Fat pointers
Consider the following example, how can we ensure that \( p \) is always in bounds?

```c
int sum(int * p, int n)
{
   int sum = 0;
   for (int i = 0; i < n; ++i)
   {
      sum += *(p+i);
   }
   return sum;
}
```

With fat pointers, each pointer \( p \) comes with bounds information. We can use these bounds to perform runtime assertions on the bounds of \( p \).

```c
int sum(int * p, int n, int * p_lo, int * p_hi)
{
   int sum = 0;
   for (int i = 0; i < n; ++i)
   {
      assert(p_lo <= p + i < p_hi);
      assert(p + i != NULL);
      sum += *(p+i);
   }
   return sum;
}
```

If \( p \) is a local variable, then the bounds variables are introduced in \( p \)’s scope (done automatically by the compiler):

```c
{
   int * p;
   int * p_lo, * p_hi;
}
```

What about assignment? When you assign pointer \( a \) to pointer \( b \) then pointer \( a \)’s bounds should be assigned to \( b \)’s bounds as well. Bounds can be determined when you `malloc` something and they can be set to zero when you free.

```c
int foo()
{
```
// On malloc...
int * p = malloc(2 * sizeof(int));
int * p_lo = p;
int * p_hi = p + 2;

// On assignment...
int * p = q;
p_lo = q_lo;
p_hi = q_hi;

// On free...
free(p);
p = 0;
p_lo = p_hi = 0;
}

On a function call, the compiler can perform the following actions automatically:

```c
int p[3];
int * p_lo = p;
int * p_hi = p + 3;
sum(p, 5, p_lo, p_hi);
```

For pointers declared within structures, bounds can be added in the structure:

```c
struct foo {
    int * a;
    int n;
    // Add bounds for a in the struct
    int * a_lo, * a_hi;
}
```

```c
struct foo * p = malloc(sizeof(struct foo));
// Introduce p_lo, p_hi for p
struct foo * p_lo = p, * p_hi = p + 1;
```

```c
p->a = malloc(2*sizeof(int));
// Bound p->a
p->a_lo = p->a;
p->a_hi = p->a + 2
```

On a function call, the bounds for p->a are already in the struct, but we also need the bounds for p:

```c
// You need bound checking for p and p->a
int funct(struct foo * p, struct foo * p_lo, struct foo * p_hi)
{
    assert(p_lo <= p < p_hi);
    assert(p != NULL);
    assert(p->a_lo <= p->a < p->a_hi);

    int tmp = *(p->a)

    return tmp + 2;
}
```

Advantages of this approach:
- Foolproof (sound)
- No programmer’s effort (the compiler inserts the bounds and the checking code)
Disadvantages of this approach:

- Performance overhead (runtime checks are performed on the bounds)
- Memory overhead (new variables are introduced to store the bounds)
- Not backwards compatible
  - Function prototypes are changed so compatibility is broken.
  - Whenever a function is instrumented, its callers have to be instrumented as well.
  - Similar for structs, you end up breaking programs
  - It cannot be applied to a library model

Reducing overhead example

Sometimes overhead can be reduced by performing some of the runtime checks, statically at compile time:

```c
struct foo * p = malloc(...);
struct foo * p_lo = p, * p_hi = p+1;

// We can statically see that this check will always evaluate to true,
// so we can get rid of the check. The compiler can do dead code elimination
// and get rid of the bounds variables itself.
assert(p_lo <= p < p_hi);
p->n = 3;
```

Deputy approach

The idea is to let the programmer specify the bounds of a pointer in terms of expressions in the same “scope.” For formal arguments, the programmer can only use other formal arguments in the expression.

With the previous approach we initially had:

```c
int sum(int * p, int n)
{
    int s = 0;
    for (int i = 0; i < n; ++i)
    {
        s += *(p+i);
    }
    return s;
}
```

And then we had the compiler perform the following changes to ensure proper bounds:

```c
int sum(int * p, int n, int * p_lo, int * p_hi)
{
    int sum = 0;
    for (int i = 0; i < n; ++i)
    {
        assert(p_lo <= p + i < p_hi);
        assert(p + i != NULL);
        sum += *(p+i);
    }
    return sum;
}
```
However, with functions like `sum`, the bounds parameters are redundant since the function already takes a size parameter `n` which can be used to infer the bounds. Therefore, instead of passing `p_lo` and `p_hi`, the programmer can specify to the compiler that the bounds of `p` are `BND(p, p + n)`.

The compiler can then generate the runtime checks like before, but now the function’s prototype is left unchanged, which provides better compatibility.

```c
int * p = malloc(2 * sizeof(int));
BND(p, p + 2);
// This is safe
sum(p, 1);
// This is not safe
sum(p, 3);
```

The rule is if `p` is a pointer such that `BND(p_{lo}, p_{hi})` and we pass it to a context which expects the bounds to be `p_{lo}', p_{hi}'`, then it must be the case that `(p_{lo}', p_{hi}') \subseteq (p_{lo}, p_{hi})` and `p` is in `[p_{lo}, p_{hi}]`.

```c
struct foo {
    int * a; // Programmer specifies BND(a, a+n)
    int n;
}

int bar(struct foo * /* BND(p, p + 1) */ p)
{
    assert(p <= p < p + 1);
    assert(p != NULL);

    int * /* BND(p->a, p->a + n) */ tmp = p->a;
    assert(p->a <= tmp < p->a + n);
    assert(tmp != NULL);
    int n = *tmp;
    return n;
}
```

Advantages:
- Backwards compatibility

Disadvantages:
- Overhead because of runtime-assertions
  - Get rid of some of them using static analysis
- Programmer overhead, since he has to write all the bounds
  - We can do type inference, like in OCaml, PHP, and determine some of the bounds

Deputy

Deputy can do type inference for local variables like `tmp` or for `malloc` calls:

```c
int bar(struct foo * /* BND(p, p + 1) */ p)
```
assert(p <= p < p + 1);
assert(p != NULL);

int /* the deputy can infer these: BND(p->a, p->a + n) */ tmp = p->a;
assert(p->a <= tmp < p->a + n);
assert(tmp != NULL);
int n = *tmp;

return n;

The programmer still has to annotate:
- struct field pointers
- function arguments
- global variables
- function variables

The deputy can infer local variables.