Suppose there are two functions (Func1 and Func3) both calling Func2 in a program. The program could be vulnerable to this. Given in the figure below, the green line shows execution as expected and red line is the possible vulnerable execution of program.

To solve this, we use Push Down Automata. Here every time a function calls, we push a number and during return the pop method is invoked and checked the validity of that return.
This is called **Context Sensitive Intrusion Detection System.**

Now in this case, too many states will have to be maintained. This number could explode depending upon the program. We cannot ask kernel each time while making a system call, because system calls are very expensive and it will slow down the system.

To solve this we maintain Logs.

**Efficient Context Sensitive IDS**
Maintain function call trace in application memory. These logs will be at known location to the monitor. This is not much, but just some extra overhead. The main thing is, that it requires modifying the program to enable extra enforcement.
Memory Safe Compilation of C

1. June’s and Kelly Method

Goal: No pointer ever goes out of bounds.

Allowed Pointer Operations:
1. *p
2. q = p + n (if p is inbound then q is not necessarily inbound)
4. q = p (if p is inbound then q is also inbound)
5. p = malloc(…)
6. free(p)
7. p = &x
8. q = (cast *)p
9. q = NULL

Method:
1. Trace bounds of all objects.
2. Check all potentially dangerous pointer operations.

Bounds tracking data structure is used.
add(p, n) //remember new object of size n at location p
(lo, hi) = lookup(p) // gives low and high address of object

To better understand this, we will use an example.
Program with vulnerability | Program without vulnerability
---|---
char *duplicate(char *p, int n) | char *duplicate(char *p, int n)
{
| {
char *q = (char *) malloc (n); | char *q = (char *) malloc (n);
add(q, n);

int i; | int i;

for(i=0; i<n; i++) | for(i=0; i<n; i++)
{
| {
*q++ = *p++; | (lo, hi) = lookup(p);
assert (lo <=p <hi);
(lo, hi) = lookup(q);
assert (lo <=q <hi);
(lo, hi) = lookup(p);
assert (lo <=p+1 <hi);
p=p+1;
(lo, hi) = lookup(q);
assert (lo <=q+1 <hi);
q=q+1;

} | }

return q-n; | return q-n;

} | }