Taint-Enhanced Policy Enforcement

In this approach, we store a tagmap for each byte in memory representing its taint. We store a 1 to represent taint and 0 otherwise.

Also, we maintain a set of policies which specify the action to be taken based on the tagmap values for a variable. These set of policies are described separately and is not part of the program.

On 32 bit machines, char tagmap[2^30]; → tagmap for each byte of memory

Consider the following code

```c
int count = 0;

void foo(void)
{
    int x,y;

    read(STDIN, &x, sizeof(x));
tagmap[&x>>2] = 0xff;  // Set the tagmap entry for x to 1 because it is filled in from STDIN. (untrusted source)

    y = x;
tagmap[&y>>2] = tagmap[&x>>2];  // Set the tagmap entry of y to entry of x.

tagstack[count++] = tagmap[&y>>2];  // While passing the argument to another function, use the tagmap stack to pass the taintedness of the variable.

    bar(y);
}

void bar(int z)
{
    int A[10];
tagmap[&z>>2] = tagstack[count--];  // Retrieve the taintedness of the variable passed via the tagstack

    if(tagmap[&z>>2])
        // instead of abort, check against the policy – Typically programs assert on finding tainted data, but here instead of abort, we check against the policy and find out what action needs to be taken.
dangerous_op(z);
}
```
Evaluation of Static/Dynamic defense
- Execution time is more
- Memory overhead is more
- Programmer effort
- Effectiveness – The policies can be changed according to the requirements and hence easy to change the behavior of the program while working with tainted data.

Why was tainting done at a byte level?

If there was a buf[] marked as tainted and int following it, when a buffer overflow occurs the 'int' is also marked as tainted.
If tainting was done at object level, say char* for the buf, then only the buf would be marked as tainted. And the integer would be marked untainted in case of a buffer overflow.
The location of the taint inside an object is important.

A simple optimization:
void foo(void)
{
    int x,y;
    char taint_y;

    read(STDIN, &x, sizeof(x));
tagmap[&x>>2] = 0xff;
y = x;
taint_y = tagmap[&x>>2]; - This way the compiler will optimize and store the taint_y in Reg.
tagstack[count++] = taint_y;
bar(y);
}

How is the tagmap value protected?

The tagmap is part of the address space of the program. In the tagmap, the entries corresponding to the tagmap table addresses are marked as unused.
If a statement like tagmap[&p>>2] during program execution points to that entry in the tagmap which were marked as unused, then a seg fault is raised.

Working with Untransformed Libraries
If untransformed libraries are used, then the tagmap has a good chance of getting corrupted.
Mark a page above and below the tagmap table as read only. This might help when working with untransformed libraries.

False Positive Scenario
- pass untainted data to tainted pointer. If we forget to mark the pointer as untainted, then we would raise a false positive.