1) Difference between:

<table>
<thead>
<tr>
<th>CQUAL</th>
<th>Metal/Meca/xg++</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) It is about the data</td>
<td>It is about the control flow.</td>
</tr>
<tr>
<td>2) It is sound</td>
<td>It is not much sound</td>
</tr>
<tr>
<td>3) It has high false positive</td>
<td>It has low false positive</td>
</tr>
<tr>
<td>4) It can find only one type of bug</td>
<td>It can find many types of bugs</td>
</tr>
<tr>
<td>5) It looks up into the whole program</td>
<td>It does local analysis</td>
</tr>
</tbody>
</table>

Q) How can we define soundness of any tool?
Ans) It means that it won't misses any bugs. And it has low no of false positive.

Q) False Negative?
Ans) It means that there is a bug but it doesn't get reported.

Q) What are the advantages and disadvantages of a Static analyser?
Ans) It requires Specification to some degree.
- It catches the bug early.
- It is language specific.
- It could be slow
- It has no run time overhead
- It has extremely high coverage.
- It reduces the testing time.
- It has low errors.
- It is system specific (like linux)

Cqual introduces two more type qualifiers to the existing type system in C. And these are tainted and untainted.

$ tainted int x;   \ this means that variable x may hold dangerous data
$ untainted int y; \ It means y can't hold dangerous types
But x = y is allowed as x may contain tainted data.

Simple rule to check whether there is any vulnerability in the code by using the above type qualifiers is that type must match. For example lets look at the following example.

int printf($untainted Char *, ....)
{
   $ tainted char *S;
   printf(s); // rejected as we are again assigning tainted data to untainted data.
}
Another example.

```c
int tprint($tainted data *); is a function

tprintf("hello")  //we are calling this function  // data get assigned hello
     //allowed as data may contain tainted data or untainted data
Cqual has type inference by which we can infer that whether the variable is tainted
or untainted. And we don't have to explicitly
assign the these type qualifier to every variable but instead to some variable and for other
variables who is not type qualified we can infer their types.
Lets look up the following example.

$ tainted int x;
$untainted int y;
y = x;
$Qz int Z   // Qz is the type qualifier that we have to infer.
Z = X;     // this means Z becomes tainted as X is also tainted.
Y = Z      // As Z is tainted and we are assigning tainted to untainted. So this violates
            the rule as Y should be untainted.

Now lets discuss in more general way.
```

For example.

```c
$ tainted int x;
$untainted int y;
$Qz int Z;

Y = Z     // it implies $Qz <= $untainted
Z = X     // It implies $tainted <= $Qz
```

So from both the above equation we imply that $tainted <= $Qz <= $untainted
So the rule is violated.

We can also check this by a graph. We can draw a graph with all variable as nodes and then
we draw directed edges according to the rule: like if $Untainted x <= $tainted int y
Then we draw directed edge from X to Y.So if there is path from tainted to untained. Then
we can infer that there is a bug.
So for the above example we can draw the following graph and as we see that there exist a
path from tained to untained variable so there is a bug.
This is also shown by the following example.
Char * strcpy(char *dst , char *src)
{
    char * temp
    while(*src){
        *dst ++ = *src++;
        return temp;
    }
}

prog1(){
    $tainted char * s;
    char * t;
    char * y = strcpy(t,s);
    printf(t);
}

So for this graph will be
So we have constructed the graph with directed edges for the above example. So we can see in the above graph that there is a path from tainted to untainted data. And this path is shown by black lines.

How can we remove such bugs. One point could be that when entering function we could sanitize the tainted variable. Like this is shown in the following example.

```c
$ untainted char * sanitize($ tainted char * t)
{
    // clean it up here ( example remove %n if it has any or %s)
    return ($untainted char *)t;
}
```

Assign the return value to printf so arguments to printf becomes untainted like we can do is printf(sanitize(t)); will always pass untainted output.

Even to prevent the cross site attack we can sanitize the output that is output should be untainted.

In this way every output to database will be untainted only hence we can prevent attacks.