==Ph4nt0m Security Team==

Issue 0x03, Phile #0x07 of 0x07

|=---=|
|=----------------------=[WEB应用安全设计思想]=-----------------------=|
|=---=|
|=---=|
|=-------------------------=[By axis   ]=-------------------------=|
|=----------------------=[<axis#ph4nt0m.org>   ]=----------------------=|
|=---=|

[目录]

一、前言

我一直在思考的一个问题，就是安全问题的本质到底是什么。我们见到过各种各样的攻击，也做过各种各样的防御方案。有的方案好，有的方案却有缺陷。那么好的方案好在哪里，为什么就能够抵抗攻击，到底什么特性使得攻击者的成本升高了，使得风险降低了。这中间是否有什么共同的东西呢？

经过一段时间的思考和观察，我初步得出了一个结论：安全问题的本质是信任问题。

二、信任关系的划分是安全设计的基础

安全问题的本质是信任问题。

提到这个，不得不说一个信任域的概念。当系统信任某些单元时，由这些单元组成的一片区域可以称之为信任域。在数据流图或者是拓跋图上，都可以用一个边界把这个域给界定出来。我说的这个概念，是一个广义的概念，任何存在信任关系的系统中，都可以存在信任域。

比如一个机场，人们要登机，必须要先经过安检，那么过了安检后，在候机厅候机，就可以把候机厅看做是一个信任域。因为对于机场来说，候机厅内的区域是可信的。而候机厅外的区域是不可信的。

机场的安检就是对跨越信任边界的一个检查。会检查有没有刀具，有没有液体、打火机等。

那么安全问题是怎么发生的呢？首先是没有合理的划分信任域，或者是信任域比较混乱。

其次就是信任边界的检查出现问题的时候。这些问题可以是检查不够充分，或者是检查没有覆盖到整个信任边界。

而这些问题导致的结果，都是产生信任危机，也就产生安全问题了。

对于传统的内存攻击来说，一个字符串超出了分配给它的指定空间长度，也可以看做是对信任域的破坏，或者是缺乏审计。

所以信任域和信任边界是非常重要的东西。在做安全方案的时候，首先就要依据资产等级，去划分信任域和信任边界。

我们要知道我们到底要保护什么东西，然后去分析有什么途径能够达到这些要保护的信任域。

在圈子里经常讲的一个笑话就是，怎么做到安全？拨网线最安全。首先，这是一个谬论，因为网线拔掉后，可用性会受到影响。安全方案应该尽可能的避免牺牲可用性为代价，应该是为业务和应用服务的。拔网线是一种舍本逐末的做法。

其次，拔了网线真的就安全了吗？

我们把物理隔绝的系统看做是一片信任域，那么它会信任什么？如何与外界做数据交互？简单的头脑风暴一下，就可以知道，这样的系统，可能会与外界发生数据交互的情况：

1. U盘有可能拷贝数据
2. 无线网卡有可能自动连接
3. 可能有人为的手工操作

那么以上这三条，都是有可能穿越我们的信任边界，产生数据流动的行为。原本物理隔绝就是为了不信任外界的一切，产生数据流动后，就可能破坏信任关系。

再回过头来看上面的机场的案例，把客流量看做是数据流量，它将穿越一道信任边界，进入候机厅这个信任域，所以机场有安检，来专门检查这个穿越信任边界的数据。安检就是机场的安全方案。

-tips----------
如果A信任B，或者A依赖于B，则B可以决定A的安全。常见的案例比如软件中使用了第三方包，则第三方包可以决定A中相关数据的安全。

某些视频播放软件使用了很多第三方的库来解析很多不同的视频格式，当第三方库出现安全问题时，则直接导致这些视频播放软件也出现安全问题。

所以安全域的划分是安全方案的基础，划分了安全域后，才能比较有针对性的设计安全方案。

三、访问控制是安全设计的核心

访问控制不仅仅包括权限。权限仅仅只是访问控制的一部分。这里我们通常所说的权限都是垂直权限控制，它一般是基于角色的（role based）。

比如一个论坛里面，有匿名用户，他们可能看不了帖子的内容。有普通用户，他们能看帖子的内容。有管理员，他们能删帖子，能置顶帖子。

那么匿名用户、普通用户、管理员就是三个不同的角色。

我们的大部分访问控制系统，都是基于角色的。普通用户没办法执行管理员的操作，因为访问控制系统会校验用户的角色，以决定他们是否有足够的权限去执行一次访问。

访问控制系统一般在整个系统中处于一个比较中心的位置，也只有让他处在一个中心的、关键的位置，才能保证每次访问都由它来控制。

但是目前我们的大多数系统都仅仅是垂直权限控制，而对水平权限控制方面却做的不太好。

什么是水平权限控制？

这个概念是相对于垂直权限控制来说的。

A与B都是同一个角色的普通用户。A上传了一个头像，系统给它编号为123，正常情况下，A可以执行“http://www.test.com/delete?id=123”去删除自己的头像。

但是由于这个删除操作仅仅校验了用户的角色，而没有校验提交该请求的用户是否是A，从而导致B可以提交以上请求，去删除A的头像。

这就是一个典型的水平权限控制出错的例子。

而很多系统中，同一个角色的用户可以加入不同的用户组，这些一个个的用户组，就是一个水平权限控制的系统。

只是问题往往出在访问控制系统的粒度上。如果划分的粒度不够细，那么一个用户组内的用户是否可以删除或修改各自的数据？

对于粒度的划分，我把一个访问控制系统中的最小单位称之为一个原子权限。无论是水平权限系统还是垂直权限系统，可能都是对原子权限的不同组合。

这个问题实际上是一个非常难以解决的问题，特别是在已经成型的大型系统中。对于现在的大型互联网公司来说，网站的代码一般都是几十G的数量级，业务系统繁多。而水平权限控制的一般要求是，将所操作的数据与用户联系起来。

回到上面的例子：delete?id=123

那么怎么知道123这条数据，是A的呢？系统无从判断，只能去查询user表。如果业务系统一复杂，可能就涉及到跨表查询或者是联合查询，甚至是跨库查询，这基本上是一场噩梦。

可是如果不进行二次查询，则无法在根本的地方解决这个问题。可是二次查询又会带来性能上的消耗。真是一个很矛盾的事情。

所以最好的做法是在设计数据层的时候，事先考虑好这个问题，做好数据与用户之间的关联性。

如果已经成型的系统，就只能在外面包一层，把这个问题隐藏起来了。在本文的后面，会提到这种做法。

除了水平和垂直权限控制外，实际上一些规则，也可以看做是访问控制。比如浏览器里的SOP（same original policy）。DOM、cookie等都有同源策略，也略有差别。但这些规则，都是

属于访问控制系统，在整个安全体系中，处于核心的位置。

-tips----------
访问控制系统一般会针对数据的RWX（读、写、执行）属性进行授权，对发起请求方则进行水平或垂直的检查。

而在WEB中，极其理想的状态，可以大胆的想象为，以session为单位建立原子权限，将数据与session关联起来后，每个不同的session就是不同的信任域，对每个跨越信任边界的请求进行水平、垂直的权限检查，这样就是一个极端理想的权限体系。

这只是一个理想模型，在实践中，需要根据实际情况进行分析。

四、数据与代码分离的思想是安全设计的原则

最典型的体现数据与代码分离思想的是模板系统。

比如velocity，在渲染html的时候，程序员可以写vm模板，一些静态写死的内容就是代码，而通过变量，经过渲染才最终展现的内容则称之为数据。一个典型的例子如下：

-code----------
```
<a href="$URI/test-$!{test.UserId}.htm" target="_blank">test</a>
```

代码与数据如果没有分离，就会导致代码混乱，数据变成代码的一部分去执行。比较常见的例子就是PHP里的SQL写法：

-code----------
```
$sql = "SELECT * FROM article WHERE articleid=".$_GET[id]."";
```

如果参数 id 中带有单引号，就会闭合掉代码中的单引号，从而导致数据变成代码执行。所以这个注射的本质问题还是没有做好数据与代码的分离。

比较好的做法是如下java代码中的使用变量绑定，很好的做到了代码与数据分离

-code----------
```
String sql="Insert into table VALUES(?,?)";
        List<Object> values = new ArrayList<Object>();
        values.add(Integer.toString(id));
        values.add(operator);
        try{
            executeStatement(sql, values);
            result=true;
        }catch(Exception e){
            e.printStackTrace();
        }
```

但是并不是说使用了模板系统就一定分离了数据与代码。

因为在类似“render”或者是“transform”的过程中，往往存在一个将数据进行规范化的过

程。这个过程也可能出现问题，从而导致代码可以混淆数据进行执行。

比较好的做法是，数据中不能包含有在代码中存在语义的字符。

参考如下例子：

```
-code---------------------------------------------------------------------------
Set-Cookie: name=id\r\nP3P: xxxxxxxxxxxxxx\r\n
----------------------------------------------------------------------------
```

红字部分是用户的输入。

在HTTP的标准中，冒号“：”，等号“=”，换行符CRLF“\r\n”，百分号“%”等字符都是有具体的语义的，属于代码部分。所以正常的用户数据中不应该包含有这些字符。

如果出于需求一定要包含怎么办？按照标准将这些字符全部encode。

在HTTP标准中可以使用urlencode，比如等号就变成了“%3d”。

这样就做到了代码与数据的分离。

代码与数据分离原则的本质还是体现了安全问题是信任问题这一思想。

代码是否应该信任数据，或者说代码应该信任怎样的数据，是这个原则的本质。

在应用中，比较好的例子是json、XSLT，这些方法都比较好的做到了数据与代码分离，所以在开发中多使用这些比较好的方法，无形中就提高了安全性。

五、最佳实践一：Secure By Default

经常可以看到一些权威文档上推荐使用“default denied”，这就是“Secure By Default”的一种体现。

“Secure By Default”可以说是一个最佳实践。在很多时候，这个思想应该上升到战略的高度。只有真正做到“Secure By Default”，才能保证网站的安全。

因为随着时间的推移和系统的发展、膨胀，会变得越来越臃肿。一个大系统发展到后期，基本上没有一个人能了解系统的全部，而变化却每天都在发生。所以，在这种情况下，只有使用“Secure By Default”的思想来制定安全方案。

白名单往往是实现“Secure By Default”的方法。与黑名单不同，白名单的思想很好的体现了“default denied”。下面以XSS的防御问题举例。

对于一些HTML的标签和事件，黑名单的做法是列出危险的标签和事件，然后禁止他们。比如列出<script>、<iframe>等标签，然后删除他们。

而白名单的做法是留下允许的，比如<a>、<img>其余的一律删除。对于白名单的做法来说，是可以抵御未知攻击的，而黑名单的做法则不行。

2009年4月，firefox 3.1 beta3更新后，开始支持HTML5里的新标准<audio>和<video>标签，而某些基于黑名单的XSS filter没有包含对这两个新标签的检查，从而出现了漏洞。而类似的漏洞在一个基于白名单的系统中是不会存在的。

一些基于异常行为检测的IPS，也很好的体现了这种“Secure By Default”的思想。比如当FTP的用户名或者密码输入非常长的时候，就认为这可能是一次攻击，因为正常的用户名和密码是不会那么长的。

这种IPS规则明显就要优于传统的基于签名匹配的IPS，因为它能探测未知攻击。

不过白名单也不是万能的，如果白名单过于宽泛，则可能让攻击隐藏在白名单中，从而绕过的系统的检查。

-tips----------
回到安全的本质问题上，白名单属于信任域，如果攻击隐藏在信任域中，就属于信任域不合理。

通配符“*”就是一个让人很头疼的东西，一定要慎用它！参考如下案例：

Flash的crossdomain.xml文件中，使用了通配符“*”，从而导致这条安全配置形同虚设。

-code----------
```
<cross-domain-policy>
<allow-access-from domain="*"/>
</cross-domain-policy>
```

所以，掌握好黑白名单的度，是一个重要的事情。

一般选择使用黑名单还是白名单，还需要考虑到工作量的问题，具体问题具体分析了。

六、最佳实践二：Unpredictable

Unpredictable可以理解为不可预测性。就是让攻击者无法预知它要攻击的目标，把我们要保护的东西藏起来！

不可预测性能够有效的对抗基于篡改、伪造的攻击。

这些攻击包括但不局限于 XSS、CSRF、Sql Injection、竞争条件、访问控制问题等等。

值得注意的是，Unpredictable仅仅只是最佳实践，但并不一定就是最好的方案。因为把问题藏起来，并非最终解决了问题，如果藏的不好，或者是有迹可循，也会导致这些防御方案失效。

因为这种思想并非“在正确的地方做正确的事情”，在实践中需要结合具体情况使用。

最典型的比如CSRF的防御，目前比较通用的方案是增加一个随机的token，这个token的本质实际上就是为了让攻击者无法准确的预测到目标URL是什么，从而也就让伪造请求的攻击无法成功的实施。

但是如果通过XSS读取了页面里的token，就使得攻击者找到了隐藏的要素，从而使得伪造请求能够成功的实施。

在现代的缓冲区溢出的防御中，ASLR（栈基址随机变化）也属于不可预测性的一种体现。

因为在内存攻击中，往往需要知道一个确切的opcode地址才能让程序按照攻击者的意图执行到shellcode，ASLR让攻击者难以找到一个确切的地址，从而 有效的防御了一些内存攻击。

但是ASLR并没有去从本质上解决程序中的缓冲区溢出问题。只是把问题藏起来了。所以当攻击者通过内存信息泄露读取到了内存中的数据，或者是找到了一个固定不变的地址时，就会让这种保护变得无效。

回到前面提到的水平权限控制问题，当我们很难去执行一个真正的解决方案时，就可以考虑使用增强方案来提供一种外部的保护。

在“Delete?id=123”中，攻击者之所以能够执行成功是因为它知道“id=123”，如果使用不可预测性的思想，让攻击者无法知道“id”是什么，就能够有效的对抗这种攻击。

当删除的操作为“delete?id=asfkjaskdjfneanef”时，攻击者也就无法通过篡改id号，来实施越权访问了。

这是另外一种解决水平权限问题的思路，仅做参考。

总的来说，不可预测性是一种对安全防护的增强，可以锦上添花，但不能雪中送炭。

七、总结

那么，到底什么才是一个好的安全方案？

好的安全方案首先肯定是一个好的应用。

好的应用有什么特征？程序员可以说出许多：稳定性、易于维护、易于调试、可扩展性、高内聚，低耦合、高性能、良好的用户体验。

这些也都应该是一个优秀的安全设计者在设计它的方案时需要考虑的问题。

此外，一个好的安全设计，还要易于查找和发现安全问题，易于配置和审计。安全系统发展到一定阶段，肯定会涉及到安全监控，一个好的安全设计，会有很大的帮助。

以上，讨论了一些安全设计中的思想和原则，概括为：

* 信任关系的划分是安全设计的基础
* 访问控制系统是安全设计的核心
* 代码与数据分离是安全设计的重要原则
* 最佳实践一：Secure By Default
* 最佳实践二：合理利用不可预测性

八、参考资料

http://cwe.mitre.org/top25/#CWE-285
http://cwe.mitre.org/data/definitions/639.html
http://cwe.mitre.org/data/definitions/566.html
http://hi.baidu.com/aullik5/blog/item/342b4c4b6a20d82409f7eff2.html

-EOF-