Python 安 全 编 程 指 南

	版 本	姓 名	时 间	备 注
创建者	V1.0	吴 友 胜	2017/03/01	创 建
修 改				

目　　录

1. SQL 注入.....3
1.1 漏洞原理.....3
1.2 漏洞场景.....3
1.3 漏洞识别.....3
1.4 漏洞防御.....3
2. XSS 漏洞.....4
2.1 漏洞原理.....4
2.2 漏洞场景.....4
2.3 漏洞识别.....5
2.4 漏洞防御.....5
3. CSRF 漏洞.....6
3.1 漏洞原理.....6
3.2 漏洞场景.....6
3.3 漏洞识别.....6
3.4 漏洞防御.....6
4. 命令注入.....7
4.1 漏洞原理.....7
4.2 漏洞场景.....7
4.3 漏洞识别.....7
4.4 漏洞防御.....7
5. 文件上传.....8
5.1 漏洞原理.....8
5.2 漏洞场景.....8
5.3 漏洞识别.....8
5.4 漏洞防御.....8
6. XXE 注入.....9
6.1 漏洞原理.....9
6.2 漏洞场景.....9
6.3 漏洞识别.....9
6.4 漏洞防御.....9

1. SQL 注入

1.1 漏洞原理

动态应用需对数据库进行查询、更新、删除等操作来实现业务逻辑数据的持久化存储，若应用没有对输入进行正确处理时，恶意输入将作为参数注入到应用对数据库的SQL操作中，导致SQL注入攻击。SQL注入漏洞可用来从数据库获取、修改、导出记录、读写文件、执行命令，甚至获取数据库管理及系统的权限。

1.2 漏洞场景

常见的场景包括：查询商品和个人信息、添加个人订单信息、更新商品信息等，漏洞常见产生原因：

■ SQL 没有验证，语句使用拼接，没使用 SQL 转义

```
sqls = "SELECT * FROM content WHERE id=%s AND email=%s" %(id, email)
rets = curs.execute(sqls)
```

1.3 漏洞识别

检查输入变量的SQL操作是否有符合性验证，是否采用了正确的编码规范（见示例）。

1.4 漏洞防御

A. 对输入变量进行符合性验证；

```
ids = request.GET.get("id", "")
emails = request.GET.get("email", "")

//检查类型和业务逻辑
if (ids > 100000):
    return false

if (strlen(emails) < 10):
    return false

//检查用户名符合规则
patternName = "^[A-Za-z0-9_-]{4,}$"
patternMobile = "^\\d{11,11}$"
patternEmail = "^[A-Za-z0-9._%-]+@[A-Za-z0-9.-]+\\.[a-zA-Z]{2,4}$"
patternIPAddress = "^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}(?:25[0-5]|2
[0-4][0-9]|[01]?[0-9][0-9]?)$"
patternIPAddress = "^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}(?:25[0-5]|2
[0-4][0-9]|[01]?[0-9][0-9]?)$"
patternUrl = "^(^http://([\\w-]+\\.)+[\\w-]+(/[\\w-./?%&=]*)?$"

if re.match(patternEmail, emails):
    return true
else:
    return false
```

B. 使用规范或框架API执行SQL操作

```
//使用规范的方法
sqls = "SELECT * FROM content WHERE id=%s AND email=%s"
rets = curs.execute(sqls, (ids, emails))

//使用框架如 Django 的 Model 方法
from blog.models import Blog
rets = Blog(id=ids, email=emails)
rets.save()

//使用框架如 Django 的 raw 方法
sqls = "SELECT * FROM content WHERE id=%s AND email=%s"
Blog.objects.raw(sqls, [ids, emails])
```

2. XSS 漏洞

2.1 漏洞原理

由于应用程序没有对输入变量实施正确的转义并输出到页面中，导致攻击者可以提交包含Javascirpt代码的输入，在程序输出时Javascript代码在浏览器中被解析执行。攻击者可以通过执行Javascript代码窃取用户信息、劫持用户的会话，控制用户的浏览器、挂马等一系列恶意攻击。

对输入变量没有做验证、输出没有做限制在python代码中示例代码如下：

```
def xss_test(request):
    test_input = request.GET.get("test_input", "")
    return HttpResponse('html content %s' %(test_input))
```

XSS漏洞的分类为：

A. 反射型：应用将用户输入直接回显给用户，不对用户的输入进行存储。

B. 存储型：应用将用户输入在数据库、文件等系统中持久化存储，并将存储数据显示在用户页面中。

C. Dom 型：DOM型指输入数据不发送到服务器端，服务端应用不处理输入，由客户端浏览器执行。

2.2 漏洞场景

XSS漏洞的产生分为需要服务端应用的交互处理和无需服务端应用的交互处理，交互处理的包括：

◆ ServerToHtml：服务端应用将用户的输入显示在HTML中。

```
<DIV STYLE="<%=style%>">
<p>Orgin html:{{ cid }}</p>
<input type="text" name="test_input" value="{{ cid }}">
<a href="http://XX.example.com/{{ cid }}">href</a>
```

◆ ServerToJs：服务端应用将用户的输入显示在Javascript代码或函数调用中。

```
<script>var x='{{ cid }}';</script>
<a href="javascript:test('{{ cid }}')">Javascript Href</a>
```

◆ ServerToJsToJs指应用将用户输入JS，函数又将用户输入通过dom写到HTML或JS。

```
<a href="javascript:test('{{ cid }}')">Javascript Href</a>
function test(argA){
    document.write(argA);
}
```

◆ Dom类型：客户端代码(javascript)直接对用户的输入进行处理，无需服务端交互处理。

```
g_a=window.location.hash.split("#")[1];
function test_html_encode(a){
    alert("input arg a : "+a);
    document.write("<p>Nothing: "+a+"</p>");
}
function test_html_attribute_encode(a){
    document.write("<p><font size='"+a+"'>testAttr</font></p>");
}
function test_jsstring(a){
    document.write("<script>var t=\""+a+"\";<\/script>");
}
function test_js_function_protocol(a){
    document.write("<a href=\"javascript:test("+a+")\">test</a><br>");
}
(function(){
    var originalUrl = window.location.href,
    toUrl = originalUrl .indexOf('#')!=-1 && originalUrl.split('#')[1];
    if(toUrl) {
        window.location.href = toUrl;
    }
})();
```

2.3 漏洞识别

A. 检查用户输入变量是否进行了类型及输入符合性验证；

B. 检查所有的输出前（所有数据来源含数据库、用户输入）是否使用了正确转义（依据场景）操作。

2.4 漏洞防御

A. 对输入变量进行符合性验证，验证是否符合应用逻辑；

```
def xss_test(request):
    name = request.GET.get("name", "")
    return render_to_response('hello.html', {'name':name})
```

B. 对输出变量【服务器端】使用正确的转义操作；

```
<{{ cid }}>
<p>Reform HtmlEncode:
Reform.HtmlEncode({{ cid }})
</p></br>
<p>Apache StringEscape:
StringEscapeUtils.escapeHtml3({{ cid }})%>
</p></br>
```

◆ 输出HTML元素

```
<% String cid=request.getParameter("cid"); %>
double_string_StringEscape_input:
<input type="text" name="test" value="StringEscapeUtils.escapeHtml3({{ cid }})">
</br>
single_string_StringEscape_html3_input:
<input type="text" name="test" value='StringEscapeUtils.escapeHtml3({{cid}})'>
</br>

//StringEscape 不会处理单引号'的转义
single_string_StringEscape_html4_input:
<input type="text" name="test" value='StringEscapeUtils.escapeHtml3({{ cid }})'>
</br>
*double_string_Reform_HtmlEncode_input:
<input type="text" name="test" value="Reform.HtmlEncode({{ cid }})">
</br>
*double_string_Reform_HtmlAttributeEncode_input:
<input type="text" name="test" value="Reform.HtmlAttributeEncode({{ cid }})">
</br>
```

◆ 输出URL地址

```
<% String cid=request.getParameter("cid"); %>
URLEncoder_href:
<a href="http://www.example.com/URLEncoder.encode({{ cid }},"gb2312")">href</a>
```

◆ 输出JS正文

```
<% String cid=request.getParameter("cid"); %>
<script>
    var y=Reform.JsString({{ cid }});
    var y='StringEscapeUtils.escapeEcmaScript({{ cid }})';
</script>
```

C. 对输出变量【客户端】使用正确的转义操作。

```
JSstring_href:
<a href="javascript:test(JsString({{ cid }}))">Javascript Href</a>
</br>
function test(argA) {
    document.write(HtmlEncode(argA));
}
```

3. CSRF 漏洞

3.1 漏洞原理

CSRF跨站伪造请求漏洞，当敏感操作接口未进行身份的二次确认(验证码、一次性token、重复输入密码)时，攻击者通过直接构造恶意页面或引用特定html元素，此页面或元素的访问将导致一次敏感操作的产生，攻击者诱导使用户直接或间接访问此页面或元素，用户以登录身份、角色执行了特定的请求并进行了敏感操作以达到攻击者的目的。CSRF漏洞可产生如转账、自动发布、修改、添加信息等敏感操作，甚至用来添加用户、修改密码等操作。漏洞常见产生原因：

A．敏感操作接口操作未进行身份的二次验证，导致CSRF攻击；

B．只允许post提交，攻击者通过JS构造表单自动执行能越过此限制；

C．检查Header或Referer，特定软件的漏洞可以伪造Header或Referer，越过此限制；

D．检查Cookie中的特定值，Cookie只要属于该网站即会发送，所以检查没有任何用途。

3.2 漏洞场景

A．个人信息页面操作：更新个人信息功能；

B．转账、订单操作：生成订单、修改订单、转账；

C．内容发布：发帖、发留言；

D．用户操作：管理员界面添加、删除、修改用户信息。

3.3 漏洞识别

检查敏感操作是否使用了验证码、一次性token,重复输入密码的方式进行用户身份的验证。错误的防范方式：只允许 post 提交；检查 Header 或 Referer；检查 Cookie 中的特定值，检查没有任何用途。

3.4 漏洞防御

原则： 应至少实现用户会话级别的token,比如用户登录成功后，生成随机字符并存储在用户会话中，在用户每次敏感操作时都应通过http参数的方式携带此值，以确保用户操作未被劫持。

A．一般操作要自动添加CSRF token，Token可以采用会话Token；

◆ 用户登录后生成token

```
user = request.GET.get("user", "");
password = request.GET.get("password", "");
capature = request.GET.get("kaptcha", "");
if validate_user(user, password, capature):
    params["login"] = true
    params["token"] = code
    return params
else:
    return None
```

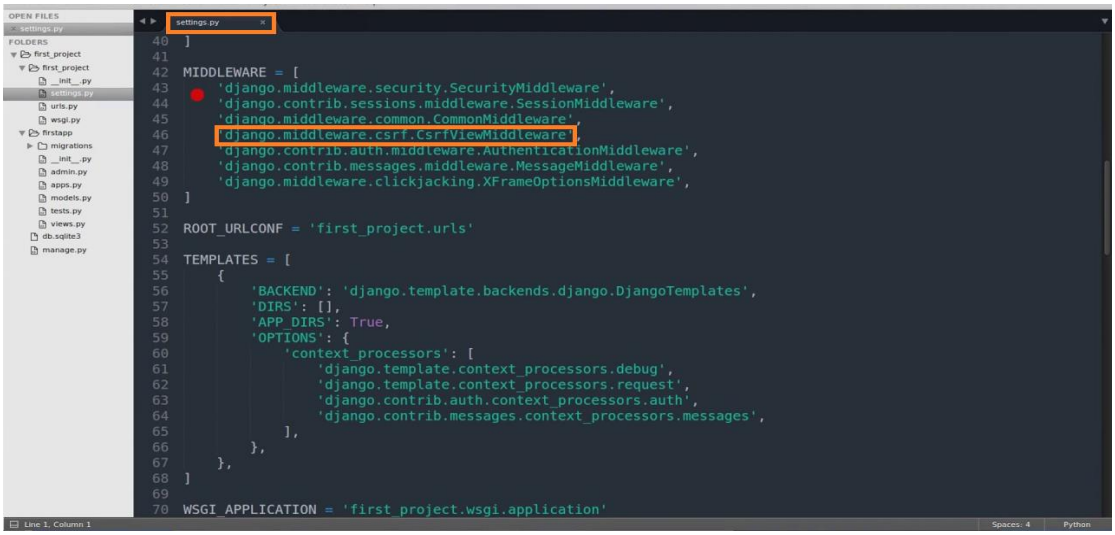
◆ 页面(模板）渲染时输出token

```
<form method=POST>
    <input name="message" type=text>
    <input type="hidden" name="token" value={{ token }}/>
    <input type="submit" value="post">
</form>
```

◆ 逻辑检查中验证token是否正确

```
if code == request.GET.get("token", ""):
    process(request)
else:
    return None
```

django 在settings.py中提供了CSRF中间件django.middleware.csrf.CsrfViewMiddleware配置即可。



在函数前加上@csrf_exempt修饰器可以实现保护。

```
from django.views.decorators.csrf import csrf_protect
from django.shortcuts import render

@csrf_protect
def my_view(request):
    c = {}
    return render(request, "a_template.html", c)
```

B. 重要操作要添加验证码或要求用户输入密码进行重复验证，如修改邮箱、转账、下单等；

4. 命令注入

4.1 漏洞原理

直接调用os.system来执行系统的shell，如果输入内容用户可控则存在注入风险，漏洞示例：

```
def myserve(request, filname, dirname):
    re = serve(request=request, path=filname, document_root=dirname)
    os.system('sudo rm -f %s' %(os.path.join(dirname, filname)))
    return re
```

备注：如果filname和dirname用户可控，则代码存在注入风险。

4.2 漏洞场景

需要根据用户输入进行系统操作如：操作文件、命令等。

4.3 漏洞识别

os.system, os.popen, os.spaw*, os.exec*, os.open, os.popen*, eval, commands.call, commands.getoutput, Popen*等。

4.4 漏洞防御

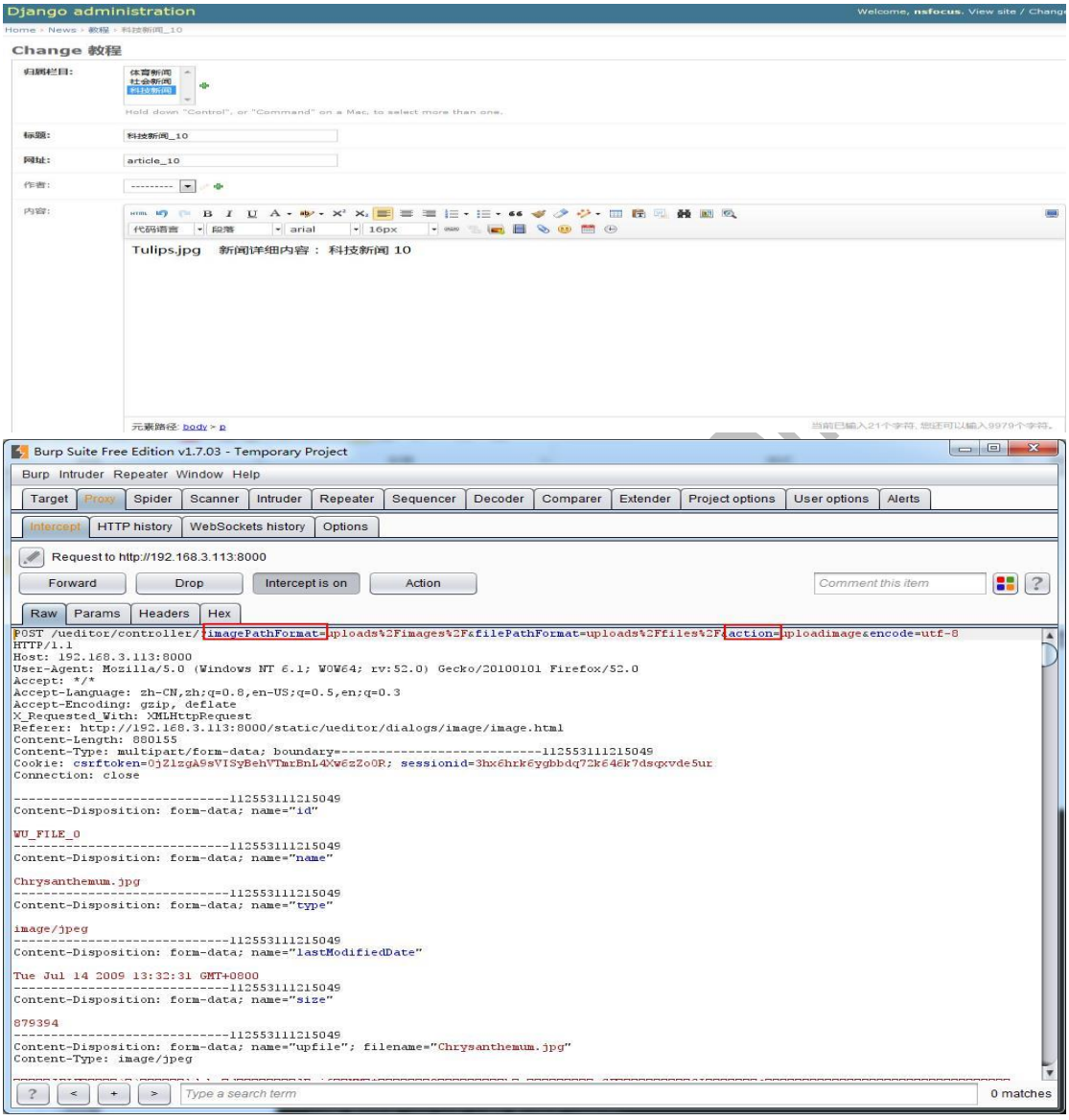
使用subprocess模块，且设置shell=False。

```
def myserve(request, filname, dirname):
    re = serve(request=request, path=filname, document_root=dirname)
    subprocess.Popen('sudo rm -f %s' %(os.path.join(dirname, filname)), shell=False)
    return re
```

5. 文件上传

5.1 漏洞原理

DjangoUeditor是一款可以在Django应用中集成百度Ueditor HTML编辑器的插件（Ueditor HTML编辑器是百度开源的在线HTML编辑器）。DjangoUeditor插件上存在一个漏洞，可以导致任意文件上传。



5.2 漏洞场景

使用了富文本编辑器Ueditor移植到Django中的组件。

5.3 漏洞识别

使用的DjangoUeditor版本为1.9.143，则存在着这个漏洞。漏洞利用就是修改imagePathFormat = uploads%2Fimages%2Fwebshell.py【默认imagePathFormat为目录】。

5.4 漏洞防御

服务器端验证 imagePathFormat 值，避免写入非uploads/images目录，或者输入为文件值。

6. XXE 注入

6.1 漏洞原理

XML文件的导出导入在libxml2.9以下可能导致xxe漏洞。

```
 [root@~ /]# cat xxe.xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE xdsec [ <!ENTITY xxe SYSTEM "file:///etc/passwd" >
]>
<root>
<node id="11" name="bb" net="192.168.0.2/24" ltd="" gid="" />test&xxe;</root>

[root@~ /]# cat xxe.py
from lxml import etree
mytree = etree.parse('xxe.xml')
etree.tostring(mytree.getroot())
```

6.2 漏洞场景

XML配置导入和导出的场合。

6.3 漏洞识别

使用了XMLParser功能。

6.4 漏洞防御

```
class XMLParser(_FeedParser)
    XMLParser(self, encoding=None, attribute_defaults=False, dtd_validation=False,
load_dtd=False, no_network=True, ns_clean=False, recover=False, XMLSchema schema=No
ne, remove_blank_text=False, resolve_entities=True, remove_comments=False, remove_p
is=False, strip_cdata=True, target=None, compact=True)
```

设置：resolve_entities=False，no_network=True，resolve_entities=False会导致解析实体失败，no_network为True阻止文件导出，但 xml.dom.minidom,xml.etree.ElementTree 不受影响。