

0x00 Unicode 简介

字符编码的发展随着图形操作系统发展加上各国语系显示的需求，从早期的 ASCII 到 EASCII 再到 ANSI 制定出代码页标准，都没有办法好好地解决字符集太小，无法同时显示全球所有字符的问题。例如：用代码页 936(GBK，简体中文)编码的文字拿去代码页 950(BIG5，繁体中文)会变成莫名其妙的文章。

这样的情况在 Unicode 编码方式(Universal Code)出现后终于得到改善，对世界上大部分的文字系统进行了整理、编码，也能支持多种语言混合的情况，是计算机科学领域里的一项业界标准。一张包含全世界所有文字的表，把所有的符号一律改成 16 进制数，没用到也用 0 来补。(注: *Unicode table*, <https://unicode-table.com/en/>)

然而，Unicode 是一个字符集，只规定了字符的二进制编码，不同系统对 Unicode 编码方式的实现方式不同。为确保传送方与接收方能收到一致的编码内容，以及出于节省空间的目的，而设计出 Unicode 转换格式(Unicode Transformation Format，简称 UTF)，常见的种类有 UTF-8、UTF-16、UTF-32、GB18030 等等。

0x01 UTF

这里简单介绍下各 Unicode 转换格式的特色，其中因 UTF-8 兼容 ASCII 且占用的空间比较小，为目前网页、Email 及网络传输应用中优先采用的编码。

UTF-8

- 一种针对 Unicode 的可变长度字符编码
- 兼容 ASCII，不需要特别转换
- 为目前网页、Email 及网络传输应用中优先采用的编码

UTF-16

- 分为 UTF-16 LE(Little-Endian)及 UTF-16 BE(Big-Endian)
- 不相容 ASCII，需转换
- 大部分字符都以固定 2 个字节来储存
- 特殊字符以 4 个字节来储存

UTF-32

- 每个字符皆固定 4 个字节来储存

补充: 微软 Windows 操作系统附带的记事本(Notepad)的另存功能中可选择四种编码, 分别是 ANSI(非 Unicode, 代码页依语言决定)、Unicode(UTF-16 LE)、Unicode big endian(UTF-16 BE)和 UTF-8

0x02 UTF-8 编码

上一章节有提到 UTF-8 采用变动长度编码, 且能兼容 ASCII, 那我们来看下他是如何编码的。UTF-8 编码会依据 Unicode 字符的区间来决定分割为多少个部份, 并分配到对应的位位置。在 U+0080 以下的字符都落于单字节编码, 刚好对应 7 位的 ASCII 字符, 其他字符则会被分割成 2~4 个不等的字节中。UTF-8 编码为了避免与 7 位的 ASCII 混淆, 每个字节的最高位皆会被设定为 1。

下面是编码表: **红色字体**表示该位值是固定, **蓝色或 xxxxx** 则为编码字符的二进制

Unicode Range	Byte 1	Byte 2	Byte 3	Byte 4
0x000000 - 0x00007F	0 xxxxxxx _(b)			
0x000080 - 0x0007FF	110 xxxxx _(b)	10 xxxxxx _(b)		
0x000800 - 0x00FFFF	1110 xxxx _(b)	10 xxxxxx _(b)	10 xxxxxx _(b)	
0x010000 - 0x1FFFFF	11110 xxx _(b)	10 xxxxxx _(b)	10 xxxxxx _(b)	10 xxxxxx _(b)

举个例子: 「\$」, U+0024 (二进制 **0100100**_(b)), 介于 0x000000-0x00007F, 使用单字节编码, 结果为 0x24 (二进制 **00100100**_(b)), 与原本 ASCII 值相同。

再个例子: 「牛」, U+725B (二进制 **01110010 01011011**_(b)), 介于 0x000800-0x00FFFF, 使用三字节编码, 依照上表将二进制依序带入 x 位置, 结果为 0xE7 0x89 0x9B(二进制 **11100111 10001001 10011011**_(b))

科普到这边, 下一章节会比较有趣一点的应用。

补充: 关于 UTF-8 详细规格的档案共有好几份, 基本原理都是相同的, 最主要的差异是支持的字符范围及无效输入的处理方法。

rfc3629 - <https://tools.ietf.org/html/rfc3629>

The Unicode Standard - <http://www.unicode.org/versions/Unicode9.0.0/ch03.pdf#G7404>

rfc2279 - <https://www.ietf.org/rfc/rfc2279.txt>

rfc2044 - <https://tools.ietf.org/html/rfc2044>

0x03 畸形字节

前一章节讲 UTF-8 的标准，但是标准归标准，实作归实作，实作与标准上的落差往往就是攻击者的利刃。那我们来试试畸形编码，不按照 Unicode Range 规则，硬把单字节编成三字节编码。

如「\$」, U+0024 (00100100_(b))编码为三字节 0xE0 0x80 0xA4(11100000 10000000 10100100_(b))

这里提供两个 UTF-8 encode/decode 功能的网址：<https://sites.google.com/site/nathanlexwww/tools/utf8-convert>、<https://mothereff.in/utf-8>。我们把刚刚编码的 0xE0 0x80 0xA4 丢进去试试。

UTF8 ⇌ Binary, Hex, and Base 64 Converter

Text

\$

Hexadecimal

E0 80 A4

Output Mode : ☐ Binary ☒ Hexadecimal ☐ RFC 4648 Base 64

UTF-8 encoder/decoder

UTF-8-decoded:
ERROR: invalid input

UTF-8-encoded: ([permalink](#))
\\xE0 \\x80 \\xA4

可以发现左边的网站即使是畸形 UTF-8 编码依然可以解出「\$」，利用这样的实作落差就有机会绕过防御机制。

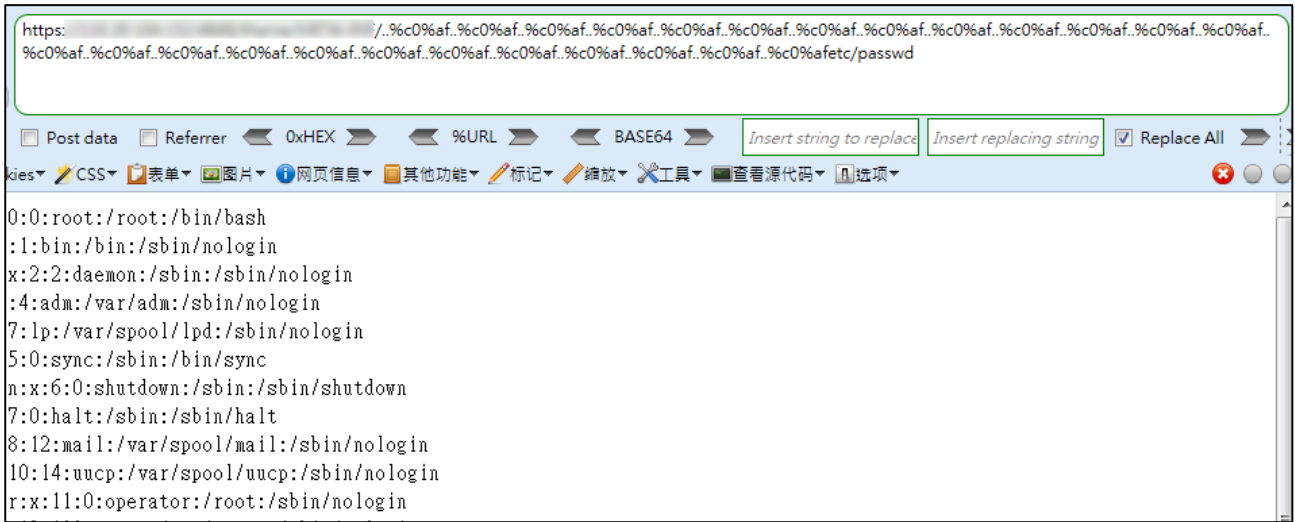
在 Oracle GlassFish Server 4.1 就因为这样的特性被找出 Directory Traversal 漏洞(根因应是更底层的 UTF-8 decode 有问题), POC 如下(<https://www.exploit-db.com/exploits/39441/>):

GET

[illegible]

(%c0%af 即为 **11000000 10101111**_(b), 「/」的双字节畸形编码)

来个实际例子



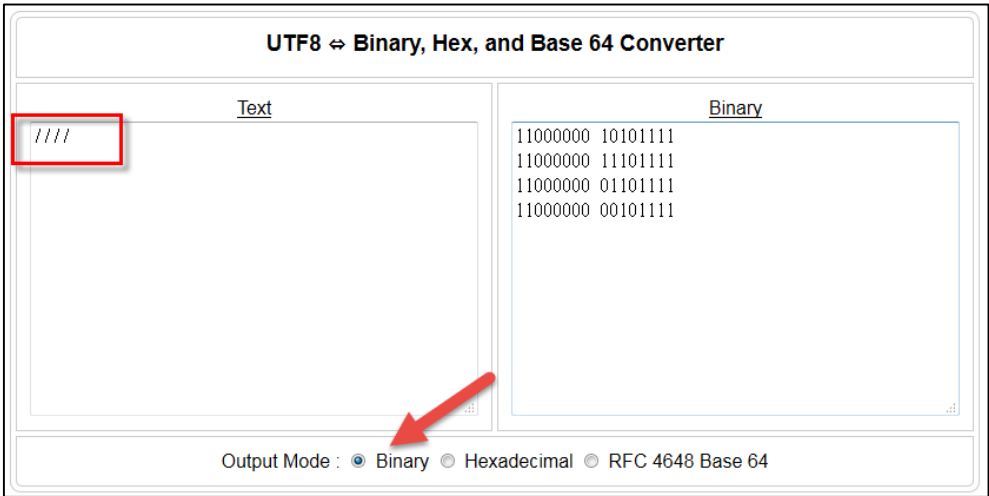
0x04 畸形检查码

前面提到的是字节上的畸形编码，这里再来看另外一种畸形编码，不填入正确的检查码。

例如「/」，U+002F(00101111_(b))即有 2 种变化 10101111、00101111，但大部分译码器对第一字节的检查码都很严谨，因此常见的作法是搭配上上述的字节畸形。

举个例子：「/」，U+002F(00101111_(b))，先转换为双字节 11000000 10101111_(b)再进行变形。

- ◆ 11000000 10101111_(b)
- ◆ 11000000 11101111_(b)
- ◆ 11000000 01101111_(b)
- ◆ 11000000 00101111_(b)



利用前面提到的译码网站，我们输入 4 个畸形编码，都能正确解出「/」字符。

那我们实际拿网站来试试，将%c0%af(11000000 10101111_(b))改为%c0%2f(11000000 00101111_(b))，毫无压力。

