

Input Normalization and Character Encodings

Document History

| Version | Date | Editor | Reviewer | Status | Remarks |
|---------|----------|-------------|----------|--------|---------|
| 1.0 | 13.03.06 | Thomas Apel | | Final | |

Created by: EUROSEC GmbH Chiffriertechnik & Sicherheit
Sodener Strasse 82 A, D-61476 Kronberg, Germany

File Name: 060125_Secologic_InputNormalization.doc

Copyright 2005, EUROSEC GmbH Chiffriertechnik & Sicherheit

Table of Content

| | | |
|-------|------------------------------------|----|
| 1 | Introduction | 4 |
| 2 | Character Encodings..... | 4 |
| 2.1 | Single-Byte Encodings..... | 5 |
| 2.1.1 | ASCII..... | 5 |
| 2.1.2 | Single-Byte ASCII Extensions | 5 |
| 2.2 | Multi-Byte Encodings for CJK | 5 |
| 2.2.1 | Shift_JIS..... | 6 |
| 2.2.2 | EUC-JP | 6 |
| 2.2.3 | Big5..... | 6 |
| 2.2.4 | GB2312..... | 7 |
| 2.3 | Unicode..... | 7 |
| 2.3.1 | UTF-8..... | 7 |
| 2.3.2 | UTF-16..... | 9 |
| 2.3.3 | UTF-32..... | 10 |
| 2.3.4 | GB18030..... | 10 |
| 3 | Problem Scenarios..... | 10 |
| 4 | Conclusion..... | 13 |
| 5 | References..... | 14 |

1 Introduction

Content filters, for example in web applications that try to filter Cross-Site Scripting code from user input, must consider that their input can come in lots of different character encodings. The most popular character encoding is probably the ASCII standard. It encodes the Latin characters used in the English language, Arabic digits, plus some special characters like punctuation marks. Other popular encodings are ISO 8859-1 and the Unicode encodings UTF-8 and UTF-16. Between these encodings the byte representation for a chosen character may vary.

This paper examines if and what security issues may arise from these encoding ambiguities. The focus of this examination is on web environments using HTML and XML. Of special interest are possible ambiguities in the encoding of characters that have special control functions in HTML or XML like the less-than sign and the greater-than sign (<, >).

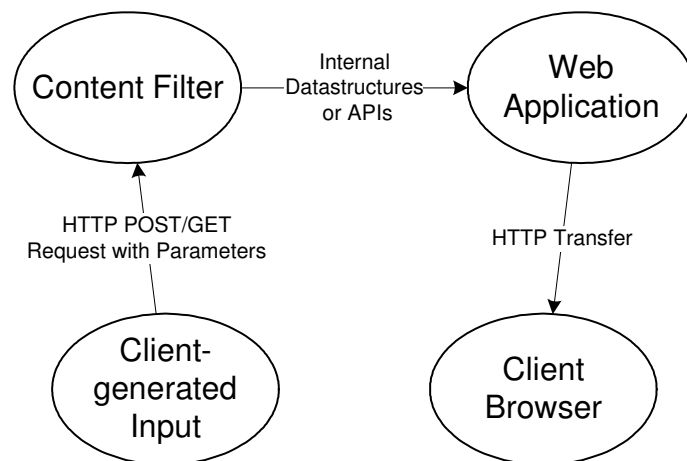


Figure 1: Overview of Involved Components

Figure 1 illustrates the software components involved in a typical scenario. In the above figure, there is a client that generates input and sends it to the web server. This is done either by a HTTP POST or GET request. On the server side there is a filter that checks the input for dangerous content. This filter passes the data to the actual web application that processes it and generates output. This output is finally delivered via HTTP to other web browsers.

The following sections present character encodings that are currently relevant in such web environments and examines possible ambiguities with respect to their relevance for content filters. In general problems may arise if different encodings use different byte representation for the same HTML control character and the content filter and the consuming software (e.g. the browser) make different assumptions on the used encoding. That way the filter might oversee harmful content because it interprets the byte stream in a different way than the consuming software.

2 Character Encodings

The following subsections present character encodings that are relevant in web environments. Section 2.1 deals with encodings that use one byte per character. This includes ASCII and its extensions that add characters specific to languages other than English. Section 2.2 presents encodings for representing characters of Asian languages like Chinese, Japanese,

and Korean (CJK). These encodings use multiple bytes for representing certain characters. The Unicode standard and its character encodings are part of section 2.3.

2.1 Single-Byte Encodings

Single-byte encodings that are relevant in web environments are 7 bit ASCII and its 8 bit extensions. Other single-byte encodings like EBCDIC are not relevant in this context.

2.1.1 ASCII

ASCII uses 7 bit codes for representing the English alphabet, Arabic digits and a couple of special characters and punctuation marks. ASCII codes representing characters are split into 8 bit bytes with their most significant bit cleared.

ASCII can be considered the native character encoding of all relevant protocols and languages in the field of web applications like HTTP and HTML. So, it serves as the standard that all other encodings must be compared to with respect to encoding differences or ambiguities.

2.1.2 Single-Byte ASCII Extensions

There are several single-byte encodings that extend ASCII with characters from languages other than English. They all use the eighth bit that is not used by ASCII for their purposes. Using this eighth bit allows to encode 128 additional characters for example German umlauts.

Examples of such 8 bit ASCII extensions are the encodings from the ISO 8859 family, DOS codepages, and Windows-125x encodings. Moreover the encodings KOI8-R and KOI8-U for Cyrillic characters, ISCII for Indian characters, and VISCII for Vietnamese characters fall into this category.

Most of these character encodings fully incorporate ASCII as they take it as a basis and only add characters by using the eighth bit. For these encodings all characters from the ASCII range are encoded in exactly the same way as they are with plain ASCII so there are no exploitable differences or ambiguities for these characters.

Exceptions are encodings like VISCII that replaces the six least problematic ASCII control characters (STX, ENQ, ACK, DC4, EM, RS) with the six least used Vietnamese characters. As these control characters have no meaning in the context of web applications this replacement does not cause any security issues.

2.2 Multi-Byte Encodings for CJK

Asian languages like Chinese, Japanese, and Korean (CJK) use more characters than can be encoded with 8 bits. The encodings presented in the following subsections are based on ASCII and extend it with multi-byte combinations for representing character from the respective language.

2.2.1 Shift_JIS

Shift_JIS extends ASCII with means for encoding Japanese characters. A basic set of Japanese characters is located in the area 0xA1-0xDF. The bytes from the ranges 0x80-0x9F and 0xE0-0xFF are lead bytes. They indicated how the following byte should be interpreted. A slight variation of Shift_JIS is Microsoft's Codepage 932 [6] that is used in Windows.

Within the ASCII range there are two changes in encoding. The byte 0x5C encodes the Yen currency symbol (¥) instead of the reverse solidus (alias backslash). The byte 0x7E encodes an overline symbol instead of the tilde.

Security problems may arise from the fact that the reverse solidus is used as directory separator in Microsoft Windows systems. Even on Japanese Windows systems that use Codepage 932 the byte 0x5C is used as directory separator. However, it is entered and displayed as a Yen symbol. System that receive Unicode data and work internally with Shift_JIS might convert both the Unicode Yen symbol (U+00A5) and the Unicode reverse solidus (U+005C) to the Shift_JIS byte 0x5C. That way both Unicode symbols will be interpreted as directory separator. If an application performs conversions in such a way any content filter that tries to protect this application must be aware of this fact as it must also treat the Unicode Yen symbol as potential directory separator. However the whole issue only applies to systems that work internally with the Shift_JIS encoding. For all other systems there is no risk if they or their filters are unaware of this issue.

Note that the same problem exists for the Taiwanese Windows codepage 949. The only difference is that it contains the Won currency symbol instead of a Yen symbol at position 0x5C.

2.2.2 EUC-JP

The EUC-JP encoding is another encoding that is widely used in Japan. It combines ASCII encoding with ISO 2022 multi-byte encoding for Japanese characters. The decision whether a byte should be interpreted as ASCII or ISO 2022 is based on the most significant bit. If the most significant bit is cleared, the byte should be interpreted as ASCII. If the most significant bit is set the byte belongs to an ISO 2022 sequence.

As a result there is no ambiguity with respect to characters from the ASCII range. Even if a system is unaware of the fact that a particular byte sequence represents EUC-JP characters, all ASCII characters will be correctly interpreted and the ISO 2022 bytes will never misinterpreted as ASCII characters as they have their most significant bit set.

2.2.3 Big5

The Big5 encoding is used for encoding traditional Chinese characters. It is a double byte character set. The first byte is in range 0xA1-0xFE, the second byte is from the ranges 0x40-0x7E and 0xA1-0xFE. It is always used in combination with single byte encoding. This is usually ASCII or one of its variants. Bytes from the range 0x00-0x80 encode the 128 characters from the ASCII range. Bytes from 0x81-0xFF either represent a character from an 8 bit ASCII extension or from the Big5 range. If there are conflicts the Big5 lead byte has precedence. This means that character from the 8 bit ASCII extension that are represented by bytes in the 0xA1-0xFE range cannot be used.

All characters from the ASCII range are encoded with the ASCII bytes. All double byte combinations that encode Chinese characters have their most significant bit set. So, all characters from the ASCII range are still interpreted correctly if Big5 encoded text is mistaken

for another encoding. This is sufficient to prevent security issues for all languages or protocols that are based on the ASCII encoding.

2.2.4 GB2312

GB2312 is another double byte encoding for Chinese characters. Besides those, it also allows encoding of Japanese, Greek, and Cyrillic characters. Moreover, it contains the ASCII range of characters. For encoding characters outside the ASCII range a lead byte from the range 0xA1-0xF7 and a second byte from the range 0xA1-0xFE are used.

Again, for ASCII characters there exists no ambiguity if a string is misinterpreted as plain ASCII or some of its variants.

2.3 Unicode

Unicode is a standard that tries to offer encodings for all possible characters from all possible languages. Each character is assigned a number, a so-called code point. These code points are usually written in hexadecimal form like U+007A for the small Latin character z. The lower 256 code points are equivalent to the ASCII extension ISO 8859-1 that is used for encoding western languages like English or German. For encoding these code points into byte sequences, there exist multiple so-called Unicode Transformation Formats (UTF). Popular Unicode Transformation Formats are UTF-8, UCS-2, UTF-16, UCS-4, UTF-32 and GB18030. The following subsections present these popular standards for encoding Unicode text.

2.3.1 UTF-8

UTF-8 uses a variable amount of bytes for encoding characters. Characters from the 7 bit ASCII range are encoded with just one byte. So UTF-8 is compatible with ASCII in a way that there is no difference between UTF-8 and ASCII as long as only characters from the 7 bit ASCII range are encoded. All ASCII encoded texts are also valid UTF-8.

For all characters outside the 7 bit ASCII range, UTF-8 uses sequences of two to four bytes for representation. Every byte from such a sequences has its most significant bit set. That way there exists no ambiguity for characters from the ASCII range, even if the processing system does not understand UTF-8 and assumes the text to be ASCII or one of its 8 bit extensions.

Another important compatibility feature of UTF-8 is that it does not contain any null bytes in contrast to other Unicode encodings like UTF-16 or UTF-32.

The following figure from RFC 3629 [1] illustrates the encoding scheme of UTF-8. Characters from the ASCII range, with code points below U+007F are encoded like they are in ASCII. They are represented by a 7 bit code, the most significant bit of each byte is cleared. Character with higher code points use two to four bytes. The first byte starts with a sequence of ones followed by a zero. The number of ones indicates the length of the byte sequence. The following bytes in the sequence have their most significant bit set to one, followed by a zero. All other bits are used for actually encoding the character. In the figure below these bits are marked with an x:

| | | |
|-------------------------------------|--|----------------------------------|
| Char. number range (hexadecimal) | | UTF-8 octet sequence (binary) |
|-------------------------------------|--|----------------------------------|

| | | | |
|------|-----------|------|-------------------------------------|
| 0000 | 0000-0000 | 007F | 0xxxxxxx |
| 0000 | 0080-0000 | 07FF | 110xxxxx 10xxxxxx |
| 0000 | 0800-0000 | FFFF | 1110xxxx 10xxxxxx 10xxxxxx |
| 0001 | 0000-0010 | FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx |

Note that not all byte sequences are valid UTF-8 sequences. Chances that text in traditional encodings like ISO 8859-x are valid UTF-8 are small. This allows UTF-8 decoders to detect if it is falsely trying to decode a ISO 8859-x encoded text.

As characters from the ASCII range use exactly the same byte representation as they do in ASCII and as all multi-byte encoded characters have their most significant bit set there is no ambiguity in the encoding of these characters that could lead to security issues.

A topic that can lead to security issues are overlong sequences. The UTF-8 encoding scheme allows to use different byte representation for the same character. For example the slash character (/) could be encoded with the following byte sequences:

| | |
|-------------|-------------------------------------|
| 2F | 00101111 |
| C0 AF | 11000000 10101111 |
| E0 80 AF | 11100000 10000000 10101111 |
| F0 80 80 AF | 11110000 10000000 10000000 10101111 |

The shortest form is encode this character with the byte 0x2F. However according to the encoding scheme the other three byte sequences would be decoded to the same value by a native decoder.

The original UTF-8 specification left some room for interpretation on how these overlong sequences should be handled. It only said that such encodings should not be produced. However how they should be decoded if they are encountered was not clarified. As a consequence many decoders treated these overlong sequences as valid representations and decoded them into the respective character. RFC 3629 and newer version of the Unicode standard clarified the handling of these overlong sequences. They must not be decoded.

However, wrong decoding of overlong sequences caused security vulnerabilities for example in Microsoft's Internet Information Server (IIS) [2] [3]. The URL filter that filters sequences like /../ that represent relative paths did not consider overlong representations. For example while it matched the hexadecimal byte sequence 0x2F 0x2E 0x2E 0x2F and recognized it as representation of /../ it did not match the overlong sequence 0x2F 0x2E 0x2E 0xC0 0xAF that represents the same string. The part of the server that performs the file system operations however decoded the overlong sequence. So as the filter and the actual string consuming function decoded the byte representation differently, it was possible to bypass the filter.

The problem of overlong sequences can be tackled at two points. In the actual string consuming system or in the upstream filter component:

- If the string consuming system does not decode overlong UTF-8 sequences, they cannot cause any harm. So consuming applications that strictly implement the UTF-8 specification sufficiently solve the problem.
- As an alternative, the problem can be solved in an upstream filter. This filter usually decodes the input string into its internal representation for processing. As long as this internal representation is not UTF-8 but for example UTF-16, it does not even harm if the filter decodes overlong sequences. During the conversion to the internal representation both the overlong sequences and the proper representation are mapped to the same internal representation. As the filter mechanism only works on this representation there is no ambiguity in the filtering process. However, it is important that only the text from the internal representation is passed on to the consuming system. The original byte sequence, possibly containing overlong sequences must not be passed.

So, in order to misuse overlong sequences two conditions must be met: The consuming system must decode overlong UTF-8 sequences while the filter is unaware of UTF-8 and does neither convert the string into an internal representation like UTF-16 nor does it filter for overlong sequences.

Note that the problem of overlong sequences is not strictly a problem of the filter but mainly a problem of the consuming software components not fully adhering to the UTF-8 specification.

2.3.2 UTF-16

UTF-16 is a variable length encoding for Unicode characters. Most characters are encoded with 16 bits. However, some require 32 bit for encoding. They are encoded by a combination of two 16 bit sequences, so called surrogate pairs.

There exists a related encoding named UCS-2. It is technically identical to UTF-16, except that it is limited to a fixed 16 bit encoding, i.e. it does not allow surrogate pairs that were introduced with Unicode 2.0. With this limitation only characters from the old Unicode 1.1 standard can be encoded. In the current Unicode standard this set of characters is called the Basic Multilingual Plane (BMP).

UTF-16 or UCS-2 are used by most Unicode aware programming languages, frameworks, and systems for internal string representation (e.g. Java, .NET, Windows NT, Mac OS Cocoa, Python).

For compatibility reasons the characters from ISO 8859-1 are encoded with their ISO 8859-1 byte sequence padded with zeros to fill 16 bits. So, the dot symbol with the ASCII byte representation 0x2E becomes the byte sequence 0x2E 0x00 in little-endian UTF-16. Encoded in big-endian UTF-16 it becomes 0x00 0x2E.

For indicating the byte order, the sequence 0xFEFF is used as Byte Order Mark (BOM) [4] at the beginning of text files. In the wrong order it reads as 0xFFFE. As this sequence does not encode any character, systems that encounter this sequence at the beginning of a file know that they have to swap bytes for decoding the text. In situations where a side channel (e.g. HTTP header) is used for indicating the byte order the character encoding names UTF-16BE and UTF-16LE are used.

When filtering and interpreting UTF-16 byte sequences, encoding ambiguities can arise from two points:

- Depending on the assumed byte order a UTF-16 byte sequence can be interpreted in two ways. Even characters from the ASCII range that are used for control purposes in languages like HTML are affected by these interpretation ambiguities.
- Characters from the ASCII range are padded with zeros. So, when filters that assume the string to be ASCII search for character sequences, they will not find them because of the interspersed null bytes. However finding single characters from the ASCII range will work fine as they use the same byte representation. The neighboring null byte padding does not prevent this.

The following example illustrates how the same byte string is treated differently depending on the byte order that is assumed. Given is the following byte sequence in hexadecimal notation:

```
3C 00 73 00 63 00 72 00 69 00 70 00 74 00 3E 00 61 00 6C 00
65 00 72 00 74 00 28 00 22 00 68 00 65 00 6C 00 6C 00 6F 00
20 00 77 00 6F 00 72 00 6C 00 64 00 21 00 22 00 29 00 3C 00
2F 00 73 00 63 00 72 00 69 00 70 00 74 00 3E 00
```

Interpreted with the proper byte order (in this case little-endian) the above byte string can be decoded to the following string:

```
<script>alert("hello world!")</script>
```

However, if the wrong byte order is assumed the same byte sequence decodes to this string of mainly Asian characters:

```
𐄀𐄁𐄂𐄃𐄄𐄅𐄆𐄇𐄈𐄉𐄊𐄋𐄌𐄍𐄎𐄏𐄐𐄑𐄒𐄓𐄔𐄕𐄖𐄗𐄘𐄙𐄚𐄛𐄜𐄝𐄞𐄟𐄠𐄡𐄢𐄣𐄤𐄥𐄦𐄧𐄨𐄩𐄪𐄫𐄬𐄭𐄮𐄯𐄰𐄱𐄲𐄳𐄴𐄵𐄶𐄷𐄸𐄹𐄺𐄻𐄼𐄽𐄾𐄿𐅀𐅁𐅂𐅃𐅄𐅅𐅆𐅇𐅈𐅉𐅊𐅋𐅌𐅍𐅎𐅏𐅐𐅑𐅒𐅓𐅔𐅕𐅖𐅗𐅘𐅙𐅚𐅛𐅜𐅝𐅞𐅟𐅠𐅡𐅢𐅣𐅤𐅥𐅦𐅧𐅨𐅩𐅪𐅫𐅬𐅭𐅮𐅯𐅰𐅱𐅲𐅳𐅴𐅵𐅶𐅷𐅸𐅹𐅺𐅻𐅼𐅽𐅾𐅿𐆀𐆁𐆂𐆃𐆄𐆅𐆆𐆇𐆈𐆉𐆊𐆋𐆌𐆍𐆎𐆏𐆐𐆑𐆒𐆓𐆔𐆕𐆖𐆗𐆘𐆙𐆚𐆛𐆜𐆝𐆞𐆟𐆠𐆡𐆢𐆣𐆤𐆥𐆦𐆧𐆨𐆩𐆪𐆫𐆬𐆭𐆮𐆯𐆰𐆱𐆲𐆳𐆴𐆵𐆶𐆷𐆸𐆹𐆺𐆻𐆼𐆽𐆾𐆿𐇀𐇁𐇂𐇃𐇄𐇅𐇆𐇇𐇈𐇉𐇊𐇋𐇌𐇍𐇎𐇏𐇐𐇑𐇒𐇓𐇔𐇕𐇖𐇗𐇘𐇙𐇚𐇛𐇜𐇝𐇞𐇟𐇠𐇡𐇢𐇣𐇤𐇥𐇦𐇧𐇨𐇩𐇪𐇫𐇬𐇭𐇮𐇯𐇰𐇱𐇲𐇳𐇴𐇵𐇶𐇷𐇸𐇹𐇺𐇻𐇼𐇽𐇾𐇿𐈀𐈁𐈂𐈃𐈄𐈅𐈆𐈇𐈈𐈉𐈊𐈋𐈌𐈍𐈎𐈏𐈐𐈑𐈒𐈓𐈔𐈕𐈖𐈗𐈘𐈙𐈚𐈛𐈜𐈝𐈞𐈟𐈠𐈡𐈢𐈣𐈤𐈥𐈦𐈧𐈨𐈩𐈪𐈫𐈬𐈭𐈮𐈯𐈰𐈱𐈲𐈳𐈴𐈵𐈶𐈷𐈸𐈹𐈺𐈻𐈼𐈽𐈾𐈿𐉀𐉁𐉂𐉃𐉄𐉅𐉆𐉇𐉈𐉉𐉊𐉋𐉌𐉍𐉎𐉏𐉐𐉑𐉒𐉓𐉔𐉕𐉖𐉗𐉘𐉙𐉚𐉛𐉜𐉝𐉞𐉟𐉠𐉡𐉢𐉣𐉤𐉥𐉦𐉧𐉨𐉩𐉪𐉫𐉬𐉭𐉮𐉯𐉰𐉱𐉲𐉳𐉴𐉵𐉶𐉷𐉸𐉹𐉺𐉻𐉼𐉽𐉾𐉿𐊀𐊁𐊂𐊃𐊄𐊅𐊆𐊇𐊈𐊉𐊊𐊋𐊌𐊍𐊎𐊏𐊐𐊑𐊒𐊓𐊔𐊕𐊖𐊗𐊘𐊙𐊚𐊛𐊜𐊝𐊞𐊟𐊠𐊡𐊢𐊣𐊤𐊥𐊦𐊧𐊨𐊩𐊪𐊫𐊬𐊭𐊮𐊯𐊰𐊱𐊲𐊳𐊴𐊵𐊶𐊷𐊸𐊹𐊺𐊻𐊼𐊽𐊾𐊿𐋀𐋁𐋂𐋃𐋄𐋅𐋆𐋇𐋈𐋉𐋊𐋋𐋌𐋍𐋎𐋏𐋐𐋑𐋒𐋓𐋔𐋕𐋖𐋗𐋘𐋙𐋚𐋛𐋜𐋝𐋞𐋟𐋠𐋡𐋢𐋣𐋤𐋥𐋦𐋧𐋨𐋩𐋪𐋫𐋬𐋭𐋮𐋯𐋰𐋱𐋲𐋳𐋴𐋵𐋶𐋷𐋸𐋹𐋺𐋻𐋼𐋽𐋾𐋿𐌀𐌁𐌂𐌃𐌄𐌅𐌆𐌇𐌈𐌉𐌊𐌋𐌌𐌍𐌎𐌏𐌐𐌑𐌒𐌓𐌔𐌕𐌖𐌗𐌘𐌙𐌚𐌛𐌜𐌝𐌞𐌟𐌠𐌡𐌢𐌣𐌤𐌥𐌦𐌧𐌨𐌩𐌪𐌫𐌬𐌭𐌮𐌯𐌰𐌱𐌲𐌳𐌴𐌵𐌶𐌷𐌸𐌹𐌺𐌻𐌼𐌽𐌾𐌿𐍀𐍁𐍂𐍃𐍄𐍅𐍆𐍇𐍈𐍉𐍊𐍋𐍌𐍍𐍎𐍏𐍐𐍑𐍒𐍓𐍔𐍕𐍖𐍗𐍘𐍙𐍚𐍛𐍜𐍝𐍞𐍟𐍠𐍡𐍢𐍣𐍤𐍥𐍦𐍧𐍨𐍩𐍪𐍫𐍬𐍭𐍮𐍯𐍰𐍱𐍲𐍳𐍴𐍵𐍶𐍷𐍸𐍹𐍺𐍻𐍼𐍽𐍾𐍿𐎀𐎁𐎂𐎃𐎄𐎅𐎆𐎇𐎈𐎉𐎊𐎋𐎌𐎍𐎎𐎏𐎐𐎑𐎒𐎓𐎔𐎕𐎖𐎗𐎘𐎙𐎚𐎛𐎜𐎝𐎞𐎟𐎠𐎡𐎢𐎣𐎤𐎥𐎦𐎧𐎨𐎩𐎪𐎫𐎬𐎭𐎮𐎯𐎰𐎱𐎲𐎳𐎴𐎵𐎶𐎷𐎸𐎹𐎺𐎻𐎼𐎽𐎾𐎿𐏀𐏁𐏂𐏃𐏄𐏅𐏆𐏇𐏈𐏉𐏊𐏋𐏌𐏍𐏎𐏏𐏐𐏑𐏒𐏓𐏔𐏕𐏖𐏗𐏘𐏙𐏚𐏛𐏜𐏝𐏞𐏟𐏠𐏡𐏢𐏣𐏤𐏥𐏦𐏧𐏨𐏩𐏪𐏫𐏬𐏭𐏮𐏯𐏰𐏱𐏲𐏳𐏴𐏵𐏶𐏷𐏸𐏹𐏺𐏻𐏼𐏽𐏾𐏿𐐀𐐁𐐂𐐃𐐄𐐅𐐆𐐇𐐈𐐉𐐊𐐋𐐌𐐍𐐎𐐏𐐐𐐑𐐒𐐓𐐔𐐕𐐖𐐗𐐘𐐙𐐚𐐛𐐜𐐝𐐞𐐟𐐠𐐡𐐢𐐣𐐤𐐥𐐦𐐧𐐨𐐩𐐪𐐫𐐬𐐭𐐮𐐯𐐰𐐱𐐲𐐳𐐴𐐵𐐶𐐷𐐸𐐹𐐺𐐻𐐼𐐽𐐾𐐿𐑀𐑁𐑂𐑃𐑄𐑅𐑆𐑇𐑈𐑉𐑊𐑋𐑌𐑍𐑎𐑏𐑐𐑑𐑒𐑓𐑔𐑕𐑖𐑗𐑘𐑙𐑚𐑛𐑜𐑝𐑞𐑟𐑠𐑡𐑢𐑣𐑤𐑥𐑦𐑧𐑨𐑩𐑪𐑫𐑬𐑭𐑮𐑯𐑰𐑱𐑲𐑳𐑴𐑵𐑶𐑷𐑸𐑹𐑺𐑻𐑼𐑽𐑾𐑿𐒀𐒁𐒂𐒃𐒄𐒅𐒆𐒇𐒈𐒉𐒊𐒋𐒌𐒍𐒎𐒏𐒐𐒑𐒒𐒓𐒔𐒕𐒖𐒗𐒘𐒙𐒚𐒛𐒜𐒝𐒞𐒟𐒠𐒡𐒢𐒣𐒤𐒥𐒦𐒧𐒨𐒩𐒪𐒫𐒬𐒭𐒮𐒯𐒰𐒱𐒲𐒳𐒴𐒵𐒶𐒷𐒸𐒹𐒺𐒻𐒼𐒽𐒾𐒿𐓀𐓁𐓂𐓃𐓄𐓅𐓆𐓇𐓈𐓉𐓊𐓋𐓌𐓍𐓎𐓏𐓐𐓑𐓒𐓓𐓔𐓕𐓖𐓗𐓘𐓙𐓚𐓛𐓜𐓝𐓞𐓟𐓠𐓡𐓢𐓣𐓤𐓥𐓦𐓧𐓨𐓩𐓪𐓫𐓬𐓭𐓮𐓯𐓰𐓱𐓲𐓳𐓴𐓵𐓶𐓷𐓸𐓹𐓺𐓻𐓼𐓽𐓾𐓿𐔀𐔁𐔂𐔃𐔄𐔅𐔆𐔇𐔈𐔉𐔊𐔋𐔌𐔍𐔎𐔏𐔐𐔑𐔒𐔓𐔔𐔕𐔖𐔗𐔘𐔙𐔚𐔛𐔜𐔝𐔞𐔟𐔠𐔡𐔢𐔣𐔤𐔥𐔦𐔧𐔨𐔩𐔪𐔫𐔬𐔭𐔮𐔯𐔰𐔱𐔲𐔳𐔴𐔵𐔶𐔷𐔸𐔹𐔺𐔻𐔼𐔽𐔾𐔿𐕀𐕁𐕂𐕃𐕄𐕅𐕆𐕇𐕈𐕉𐕊𐕋𐕌𐕍𐕎𐕏𐕐𐕑𐕒𐕓𐕔𐕕𐕖𐕗𐕘𐕙𐕚𐕛𐕜𐕝𐕞𐕟𐕠𐕡𐕢𐕣𐕤𐕥𐕦𐕧𐕨𐕩𐕪𐕫𐕬𐕭𐕮𐕯𐕰𐕱𐕲𐕳𐕴𐕵𐕶𐕷𐕸𐕹𐕺𐕻𐕼𐕽𐕾𐕿𐖀𐖁𐖂𐖃𐖄𐖅𐖆𐖇𐖈𐖉𐖊𐖋𐖌𐖍𐖎𐖏𐖐𐖑𐖒𐖓𐖔𐖕𐖖𐖗𐖘𐖙𐖚𐖛𐖜𐖝𐖞𐖟𐖠𐖡𐖢𐖣𐖤𐖥𐖦𐖧𐖨𐖩𐖪𐖫𐖬𐖭𐖮𐖯𐖰𐖱𐖲𐖳𐖴𐖵𐖶𐖷𐖸𐖹𐖺𐖻𐖼𐖽𐖾𐖿𐗀𐗁𐗂𐗃𐗄𐗅𐗆𐗇𐗈𐗉𐗊𐗋𐗌𐗍𐗎𐗏𐗐𐗑𐗒𐗓𐗔𐗕𐗖𐗗𐗘𐗙𐗚𐗛𐗜𐗝𐗞𐗟𐗠𐗡𐗢𐗣𐗤𐗥𐗦𐗧𐗨𐗩𐗪𐗫𐗬𐗭𐗮𐗯𐗰𐗱𐗲𐗳𐗴𐗵𐗶𐗷𐗸𐗹𐗺𐗻𐗼𐗽𐗾𐗿𐘀𐘁𐘂𐘃𐘄𐘅𐘆𐘇𐘈𐘉𐘊𐘋𐘌𐘍𐘎𐘏𐘐𐘑𐘒𐘓𐘔𐘕𐘖𐘗𐘘𐘙𐘚𐘛𐘜𐘝𐘞𐘟𐘠𐘡𐘢𐘣𐘤𐘥𐘦𐘧𐘨𐘩𐘪𐘫𐘬𐘭𐘮𐘯𐘰𐘱𐘲𐘳𐘴𐘵𐘶𐘷𐘸𐘹𐘺𐘻𐘼𐘽𐘾𐘿𐙀𐙁𐙂𐙃𐙄𐙅𐙆𐙇𐙈𐙉𐙊𐙋𐙌𐙍𐙎𐙏𐙐𐙑𐙒𐙓𐙔𐙕𐙖𐙗𐙘𐙙𐙚𐙛𐙜𐙝𐙞𐙟𐙠𐙡𐙢𐙣𐙤𐙥𐙦𐙧𐙨𐙩𐙪𐙫𐙬𐙭𐙮𐙯𐙰𐙱𐙲𐙳𐙴𐙵𐙶𐙷𐙸𐙹𐙺𐙻𐙼𐙽𐙾𐙿𐚀𐚁𐚂𐚃𐚄𐚅𐚆𐚇𐚈𐚉𐚊𐚋𐚌𐚍𐚎𐚏𐚐𐚑𐚒𐚓𐚔𐚕𐚖𐚗𐚘𐚙𐚚𐚛𐚜𐚝𐚞𐚟𐚠𐚡𐚢𐚣𐚤𐚥𐚦𐚧𐚨𐚩𐚪𐚫𐚬𐚭𐚮𐚯𐚰𐚱𐚲𐚳𐚴𐚵𐚶𐚷𐚸𐚹𐚺𐚻𐚼𐚽𐚾𐚿𐛀𐛁𐛂𐛃𐛄𐛅𐛆𐛇𐛈𐛉𐛊𐛋𐛌𐛍𐛎𐛏𐛐𐛑𐛒𐛓𐛔𐛕𐛖𐛗𐛘𐛙𐛚𐛛𐛜𐛝𐛞𐛟𐛠𐛡𐛢𐛣𐛤𐛥𐛦𐛧𐛨𐛩𐛪𐛫𐛬𐛭𐛮𐛯𐛰𐛱𐛲𐛳𐛴𐛵𐛶𐛷𐛸𐛹𐛺𐛻𐛼𐛽𐛾𐛿𐜀𐜁𐜂𐜃𐜄𐜅𐜆𐜇𐜈𐜉𐜊𐜋𐜌𐜍𐜎𐜏𐜐𐜑𐜒𐜓𐜔𐜕𐜖𐜗𐜘𐜙𐜚𐜛𐜜𐜝𐜞𐜟𐜠𐜡𐜢𐜣𐜤𐜥𐜦𐜧𐜨𐜩𐜪𐜫𐜬𐜭𐜮𐜯𐜰𐜱𐜲𐜳𐜴𐜵𐜶𐜷𐜸𐜹𐜺𐜻𐜼𐜽𐜾𐜿𐝀𐝁𐝂𐝃𐝄𐝅𐝆𐝇𐝈𐝉𐝊𐝋𐝌𐝍𐝎𐝏𐝐𐝑𐝒𐝓𐝔𐝕𐝖𐝗𐝘𐝙𐝚𐝛𐝜𐝝𐝞𐝟𐝠𐝡𐝢𐝣𐝤𐝥𐝦𐝧𐝨𐝩𐝪𐝫𐝬𐝭𐝮𐝯𐝰𐝱𐝲𐝳𐝴𐝵𐝶𐝷𐝸𐝹𐝺𐝻𐝼𐝽𐝾𐝿𐞀𐞁𐞂𐞃𐞄𐞅𐞆𐞇𐞈𐞉𐞊𐞋𐞌𐞍𐞎𐞏𐞐𐞑𐞒𐞓𐞔𐞕𐞖𐞗𐞘𐞙𐞚𐞛𐞜𐞝𐞞𐞟𐞠𐞡𐞢𐞣𐞤𐞥𐞦𐞧𐞨𐞩𐞪𐞫𐞬𐞭𐞮𐞯𐞰𐞱𐞲𐞳𐞴𐞵𐞶𐞷𐞸𐞹𐞺𐞻𐞼𐞽𐞾𐞿𐟀𐟁𐟂𐟃𐟄𐟅𐟆𐟇𐟈𐟉𐟊𐟋𐟌𐟍𐟎𐟏𐟐𐟑𐟒𐟓𐟔𐟕𐟖𐟗𐟘𐟙𐟚𐟛𐟜𐟝𐟞𐟟𐟠𐟡𐟢𐟣𐟤𐟥𐟦𐟧𐟨𐟩𐟪𐟫𐟬𐟭𐟮𐟯𐟰𐟱𐟲𐟳𐟴𐟵𐟶𐟷𐟸𐟹𐟺𐟻𐟼𐟽𐟾𐟿𐠀𐠁𐠂𐠃𐠄𐠅𐠆𐠇𐠈𐠉𐠊𐠋𐠌𐠍𐠎𐠏𐠐𐠑𐠒𐠓𐠔𐠕𐠖𐠗𐠘𐠙𐠚𐠛𐠜𐠝𐠞𐠟𐠠𐠡𐠢𐠣𐠤𐠥𐠦𐠧𐠨𐠩𐠪𐠫𐠬𐠭𐠮𐠯𐠰𐠱𐠲𐠳𐠴𐠵𐠶𐠷𐠸𐠹𐠺𐠻𐠼𐠽𐠾𐠿𐡀𐡁𐡂𐡃𐡄𐡅𐡆𐡇𐡈𐡉𐡊𐡋𐡌𐡍𐡎𐡏𐡐𐡑𐡒𐡓𐡔𐡕𐡖𐡗𐡘𐡙𐡚𐡛𐡜𐡝𐡞𐡟𐡠𐡡𐡢𐡣𐡤𐡥𐡦𐡧𐡨𐡩𐡪𐡫𐡬𐡭𐡮𐡯𐡰𐡱𐡲𐡳𐡴𐡵𐡶𐡷𐡸𐡹𐡺𐡻𐡼𐡽𐡾𐡿𐢀𐢁𐢂𐢃𐢄𐢅𐢆𐢇𐢈𐢉𐢊𐢋𐢌𐢍𐢎𐢏𐢐𐢑𐢒𐢓𐢔𐢕𐢖𐢗𐢘𐢙𐢚𐢛𐢜𐢝𐢞𐢟𐢠𐢡𐢢𐢣𐢤𐢥𐢦𐢧𐢨𐢩𐢪𐢫𐢬𐢭𐢮𐢯𐢰𐢱𐢲𐢳𐢴𐢵𐢶𐢷𐢸𐢹𐢺𐢻𐢼𐢽𐢾𐢿𐣀𐣁𐣂𐣃𐣄𐣅𐣆𐣇𐣈𐣉𐣊𐣋𐣌𐣍𐣎𐣏𐣐𐣑𐣒𐣓𐣔𐣕𐣖𐣗𐣘𐣙𐣚𐣛𐣜𐣝𐣞𐣟𐣠𐣡𐣢𐣣𐣤𐣥𐣦𐣧𐣨𐣩𐣪𐣫𐣬𐣭𐣮𐣯𐣰𐣱𐣲𐣳𐣴𐣵𐣶𐣷𐣸𐣹𐣺𐣻𐣼𐣽𐣾𐣿𐤀𐤁𐤂𐤃𐤄𐤅𐤆𐤇𐤈𐤉𐤊𐤋𐤌𐤍𐤎𐤏𐤐𐤑𐤒𐤓𐤔𐤕𐤖𐤗𐤘𐤙𐤚𐤛𐤜𐤝𐤞𐤟𐤠𐤡𐤢𐤣𐤤𐤥𐤦𐤧𐤨𐤩𐤪𐤫𐤬𐤭𐤮𐤯𐤰𐤱𐤲𐤳𐤴𐤵𐤶𐤷𐤸𐤹𐤺𐤻𐤼𐤽𐤾𐤿𐥀𐥁𐥂𐥃𐥄𐥅𐥆𐥇𐥈𐥉𐥊𐥋𐥌𐥍𐥎𐥏𐥐𐥑𐥒𐥓𐥔𐥕𐥖𐥗𐥘𐥙𐥚𐥛𐥜𐥝𐥞𐥟𐥠𐥡𐥢𐥣𐥤𐥥𐥦𐥧𐥨𐥩𐥪𐥫𐥬𐥭𐥮𐥯𐥰𐥱𐥲𐥳𐥴𐥵𐥶𐥷𐥸𐥹𐥺𐥻𐥼𐥽𐥾𐥿𐦀𐦁𐦂𐦃𐦄𐦅𐦆𐦇𐦈𐦉𐦊𐦋𐦌𐦍𐦎𐦏𐦐𐦑𐦒𐦓𐦔𐦕𐦖𐦗𐦘𐦙𐦚𐦛𐦜𐦝𐦞𐦟𐦠𐦡𐦢𐦣𐦤𐦥𐦦𐦧𐦨𐦩𐦪𐦫𐦬𐦭𐦮𐦯𐦰𐦱𐦲𐦳𐦴𐦵𐦶𐦷𐦸𐦹𐦺𐦻𐦼𐦽𐦾𐦿𐧀𐧁𐧂𐧃𐧄𐧅𐧆𐧇𐧈𐧉𐧊𐧋𐧌𐧍𐧎𐧏𐧐𐧑𐧒𐧓𐧔𐧕𐧖𐧗𐧘𐧙𐧚𐧛𐧜𐧝𐧞𐧟𐧠𐧡𐧢𐧣𐧤𐧥𐧦𐧧𐧨𐧩𐧪𐧫𐧬𐧭𐧮𐧯𐧰𐧱𐧲𐧳𐧴𐧵𐧶𐧷𐧸𐧹𐧺𐧻𐧼𐧽𐧾𐧿𐨀𐨁𐨂𐨃𐨄𐨅𐨆𐨇𐨈𐨉𐨊𐨋𐨌𐨍𐨎𐨏𐨐𐨑𐨒𐨓𐨔𐨕𐨖𐨗𐨘𐨙𐨚𐨛𐨜𐨝𐨞𐨟𐨠𐨡𐨢𐨣𐨤𐨥𐨦𐨧𐨨𐨩𐨪𐨫𐨬𐨭𐨮𐨯𐨰𐨱𐨲𐨳𐨴𐨵𐨶𐨷𐨹𐨺𐨸𐨻𐨼𐨽𐨾𐨿𐩀𐩁𐩂𐩃𐩄𐩅𐩆𐩇𐩈𐩉𐩊𐩋𐩌𐩍𐩎𐩏𐩐𐩑𐩒𐩓𐩔𐩕𐩖𐩗𐩘𐩙𐩚𐩛𐩜𐩝𐩞𐩟𐩠𐩡𐩢𐩣𐩤𐩥𐩦𐩧𐩨𐩩𐩪𐩫𐩬𐩭𐩮𐩯𐩰𐩱𐩲𐩳𐩴𐩵𐩶𐩷𐩸𐩹𐩺𐩻𐩼𐩽𐩾𐩿𐪀𐪁𐪂𐪃𐪄𐪅𐪆𐪇𐪈𐪉𐪊𐪋𐪌𐪍𐪎𐪏𐪐𐪑𐪒𐪓𐪔𐪕𐪖𐪗𐪘𐪙𐪚𐪛𐪜𐪝𐪞𐪟𐪠𐪡𐪢𐪣𐪤𐪥𐪦𐪧𐪨𐪩𐪪𐪫𐪬𐪭𐪮𐪯𐪰𐪱𐪲𐪳𐪴𐪵𐪶𐪷𐪸𐪹𐪺𐪻𐪼𐪽𐪾𐪿𐫀𐫁𐫂𐫃𐫄𐫅𐫆𐫇𐫈𐫉𐫊𐫋𐫌𐫍𐫎𐫏𐫐𐫑𐫒𐫓𐫔𐫕𐫖𐫗𐫘𐫙𐫚𐫛𐫜𐫝𐫞𐫟𐫠𐫡𐫢𐫣𐫤𐫦𐫥𐫧𐫨𐫩𐫪𐫫𐫬𐫭𐫮𐫯𐫰𐫱𐫲𐫳𐫴𐫵𐫶𐫷𐫸𐫹𐫺𐫻𐫼𐫽𐫾𐫿𐬀𐬁𐬂𐬃𐬄𐬅𐬆𐬇𐬈𐬉𐬊𐬋𐬌𐬍𐬎𐬏𐬐𐬑𐬒𐬓𐬔𐬕𐬖𐬗𐬘𐬙𐬚𐬛𐬜𐬝𐬞𐬟𐬠𐬡𐬢𐬣𐬤𐬥𐬦𐬧𐬨𐬩𐬪𐬫𐬬𐬭𐬮𐬯𐬰𐬱𐬲𐬳𐬴𐬵𐬶𐬷𐬸𐬹𐬺𐬻𐬼𐬽𐬾𐬿𐭀𐭁𐭂𐭃𐭄𐭅𐭆𐭇𐭈𐭉𐭊𐭋𐭌𐭍𐭎𐭏𐭐𐭑𐭒𐭓𐭔𐭕𐭖𐭗𐭘𐭙𐭚𐭛𐭜𐭝𐭞𐭟𐭠𐭡𐭢𐭣𐭤𐭥𐭦𐭧𐭨𐭩𐭪𐭫𐭬𐭭𐭮𐭯𐭰𐭱𐭲𐭳𐭴𐭵𐭶𐭷𐭸𐭹𐭺𐭻𐭼𐭽𐭾𐭿𐮀𐮁𐮂𐮃𐮄𐮅𐮆𐮇𐮈𐮉𐮊𐮋𐮌𐮍𐮎𐮏𐮐𐮑𐮒𐮓𐮔𐮕𐮖𐮗𐮘𐮙𐮚𐮛𐮜𐮝𐮞𐮟𐮠𐮡𐮢𐮣𐮤𐮥𐮦𐮧𐮨𐮩𐮪𐮫𐮬𐮭𐮮𐮯𐮰𐮱𐮲𐮳𐮴𐮵𐮶𐮷𐮸𐮹𐮺𐮻𐮼𐮽𐮾𐮿𐯀𐯁𐯂𐯃𐯄𐯅𐯆𐯇𐯈𐯉𐯊𐯋𐯌𐯍𐯎𐯏𐯐𐯑𐯒𐯓𐯔𐯕𐯖𐯗𐯘𐯙𐯚𐯛𐯜𐯝𐯞𐯟𐯠𐯡𐯢𐯣𐯤𐯥𐯦𐯧𐯨𐯩𐯪𐯫𐯬𐯭𐯮𐯯𐯰𐯱𐯲𐯳𐯴𐯵𐯶𐯷𐯸𐯹𐯺𐯻𐯼𐯽𐯾𐯿𐰀𐰁𐰂𐰃𐰄𐰅𐰆𐰇𐰈𐰉𐰊𐰋𐰌𐰍𐰎𐰏𐰐𐰑𐰒𐰓𐰔𐰕𐰖𐰗𐰘𐰙𐰚𐰛𐰜𐰝𐰞𐰟𐰠𐰡𐰢𐰣𐰤𐰥𐰦𐰧𐰨𐰩𐰪𐰫𐰬𐰭𐰮𐰯𐰰𐰱𐰲𐰳𐰴𐰵𐰶𐰷𐰸𐰹𐰺𐰻𐰼𐰽𐰾𐰿𐱀𐱁𐱂𐱃𐱄𐱅𐱆𐱇𐱈𐱉𐱊𐱋𐱌𐱍𐱎𐱏𐱐𐱑𐱒𐱓𐱔𐱕𐱖𐱗𐱘𐱙𐱚𐱛𐱜𐱝𐱞𐱟𐱠𐱡𐱢𐱣𐱤𐱥𐱦𐱧𐱨𐱩𐱪𐱫𐱬𐱭𐱮𐱯𐱰𐱱𐱲𐱳𐱴𐱵𐱶𐱷𐱸𐱹𐱺𐱻𐱼𐱽𐱾𐱿𐲀𐲁𐲂𐲃𐲄𐲅𐲆𐲇𐲈𐲉𐲊𐲋𐲌𐲍𐲎𐲏𐲐𐲑𐲒𐲓𐲔𐲕𐲖𐲗𐲘𐲙𐲚𐲛𐲜𐲝𐲞𐲟𐲠𐲡𐲢𐲣𐲤𐲥𐲦𐲧𐲨𐲩𐲪𐲫𐲬𐲭𐲮𐲯𐲰𐲱𐲲𐲳𐲴𐲵𐲶𐲷𐲸𐲹𐲺𐲻𐲼𐲽𐲾𐲿𐳀𐳁𐳂𐳃𐳄𐳅𐳆𐳇𐳈𐳉𐳊𐳋𐳌𐳍𐳎𐳏𐳐𐳑𐳒𐳓𐳔𐳕𐳖𐳗𐳘𐳙𐳚
```

input is encoded in ASCII or ISO 8859-1, can still detect the characters they want to filter as long as they ignore the null bytes and do not treat them as string terminators. They are even able to substitute these characters with other characters from the ISO 8859-1 range.

- Text encoded in an encoding other than UTF-16/32 that is treated as ASCII. This includes text encoded in UTF-8, Shift_JIS, EUC-JP, Big5, GB2312, and the ISO 8859-x family. As these encodings are ASCII supersets even filters that are ignorant of these encodings and assume the input to be ASCII are able to filter both single characters and character sequences.

This means that systems that are ignorant of character encodings and Unicode and still assume every text to be ASCII-encoded do not need to fear any security issues as long as the text is treated as ASCII throughout the whole processing chain. Note that for web applications this includes the client web browser that must assume the text to be ASCII encoded. It must not interpret this text as UTF-16 or UTF-32 because of any automatic character set detection mechanisms. Details about these mechanisms are discussed below.

In contrast to the scenarios above, the following scenarios must be considered dangerous as they might allow to bypass the filter:

- Unicode text that is filtered as Unicode but is converted to Shift_JIS in the consuming system with a mapping from U+00A5 (Yen symbol) to 0x5C which lives a double life as Yen symbol and directory separator (backslash substitute) on Microsoft Windows systems. Obviously this scenario is limited to systems that use the Japanese encoding Shift_JIS for their internal string representation and convert U+00A5 to 0x5C and use an filter that is unaware the implications of this conversion. So for most web applications, especially outside Japan, this scenario is irrelevant.
- Overlong UTF-8 sequences in combination with an unaware filter and a consuming system that decodes and processes these sequences. So, this scenario requires both a filter that is either unaware of UTF-8 or cannot properly handle overlong sequences and a consuming system that also does not properly handle overlong sequences. The likelihood that these conditions are met is reduced as the consuming systems in the web context are usually web browsers and their current versions can properly handle overlong sequences. Nevertheless, filters should be aware of overlong sequences and should not allow them in user input.
- UTF-16/32 text that is interpreted as ASCII by a filter that searches for character sequences. Because of the interspersed null byte paddings, the filter will not match the search sequence. If the consumer system interprets the text as UTF-16/32 it might execute the code the filter missed. The prerequisites for this to happen are discussed below in this section.
- ASCII encoded text that is treated as UTF-16/32. The input is encoded in ASCII but the filter interprets it as UTF-16/32 and filters accordingly. Characters like < or > are not recognized. They are evaluated with their neighboring bytes into some other character. If the consuming system interprets the text as ASCII it is possible to bypass the filter as filter and the consuming system interpret the input differently. Again, the prerequisites for attacks that exploit this are discussed below in this section.
- UTF-16/32 encoded text that is interpreted by the filter and the consuming system in different byte orders. This could happen if the text comes with contradicting content type information and BOM. Another possibility is that one system respects the attached byte order information while the other ignores it and uses its native byte order which might be different. In both cases the filter and the consuming component work on different interpretations of the same byte sequence which allows to bypass the filter. Details are discussed below together with the other UTF-16/32 related scenarios.

From these scenarios especially the last three appear dangerous and relevant for real world conditions. However, they require an attacker to influence the way the systems further down the processing chain interpret the submitted byte string. The following paragraphs give a short overview of how character encodings are handled in the context of HTTP and web forms.

Parameters to GET requests cannot carry any information about the used encoding. Web browsers usually send them in the same encoding the page with the web form was encoded in. So for parameters to GET requests web applications can only assume that the received data is in the same encoding as the form. Other reliable means for determining the encoding scheme do not exist.

With POST requests the situation is slightly different. Their encoding can be indicated with the HTTP content type header. Unfortunately poorly-written server-side scripts got, and sometimes still get confused if the content type header contained information about the used character encoding. So, browsers developers decided not to include this information which means that web application can still only assume that the browser used for its submission the encoding of the web form.

Another possibility is to submit data in a HTTP POST request with a MIME multipart message. MIME multipart messages can carry information about the used encoding. However, again because of poorly-written applications, browser include this information only if it differs from the encoding of the page containing the form.

The only mechanism that is somewhat usable in the real world is a protocol extension introduced by Microsoft. If the form contains a hidden field with the name “_charset_”, the browser submits the name of the used character encoding as value. This feature is also implemented in Mozilla and is part of the current working draft for the Web Forms 2.0 specification.

Because of the shortcomings of the different mechanisms, most web applications simply assume that the submitted encoding matches that of the web form. As a result, an attacker usually has no possibility to influence the way his submitted byte stream is processed by the filter or the web application.

The other interesting system in the processing chain is the web browser that interprets the HTML page that might contain data submitted by an attacker. The HTTP protocol allows the web application to specify the used character encoding in the header. However, the HTTP header can usually not be influenced by an attacker. So usually the only chance for an attacker to influence the way the web browser interprets the page is to instrument browser mechanisms for automatically detecting the character encoding used by a web page.

For example the Microsoft Internet Explorer comes with such an automatic detection mechanism that is turned on by default. The used algorithm tries to guess the right encoding by analyzing the distribution and usage frequency of certain byte patterns in the web page. But the exploitability of this feature seems to be limited as it does not seem to be possible to persuade the browser to interpret text delivered as ASCII or ISO 8859-1 as UTF-16/32 by submitting the right byte patterns into the web page. The other way around, tricking the browser into thinking a UTF-16/32 page is actually ASCII, does not seem to work either.

This can only be achieved with a BOM. When the HTML file starts with a UTF-16 BOM it is interpreted as UTF-16 no matter what the HTTP header says. Similarly big-endian documents get interpreted as little-endian if the BOM is swapped no matter what the HTTP header says. However, these mechanisms are usually not available for an attacker as they require to control the first two bytes of the delivered page.

With Mozilla Firefox the situation is similar. It also comes with a mechanism for automatically detecting the used encoding. The difference is that this mechanism is turned off by default. So, if present, only the HTTP header or the equivalent meta tag in the HTML page can

influence what encoding scheme is used for interpreting the delivered byte string. This behavior conforms to the HTML 4.01 specification (section 5.2.2) [5] that demands clients to use the encoding specified in the HTTP header if present. With Mozilla Firefox, the only exception are UTF-16/32 files with BOM. The BOM has precedence over the HTTP content type header. But as discussed above, the BOM usually cannot be altered by an attacker.

For the last three scenarios from the above list, this means the following:

- The scenario with UTF-16/32 text that is treated as ASCII by the filter but as UTF-16/32 by the rest of the processing chain, including the client web browser, could occur in the real world. It would mean that a web application that delivers content as UTF-16/32 is coupled with a separate filter software that does not know about Unicode and UTF-16/32 or that treats the input as ASCII because of a misconfiguration. Chances that such a combination exists in the real world can be considered low as UTF-16/32 is only scarcely used for encoding web pages.
- The next scenario, ASCII text that is interpreted as UTF-16/32 by the filter but as ASCII by the browser will usually not occur in practice. As the text is delivered as ASCII to the browser, the application will assume that all input is in ASCII, too. So, for an attacker there will be no possibility to trick the filter into treating the submitted text as UTF-16/32. The only exception would be a filter that tries to detect the encoding of the submitted data and switches to UTF-16/32 mode if it encounters a byte string with interspersed null bytes. However, this can be considered unlikely.
- Attacks based on the byte order of UTF-16/32 do not seem feasible as in practice an attacker has neither the possibility to indicate a byte order when for the submitted data nor can he influence the byte order mark when the data is delivered to the web browser.

So, there exist a couple of scenarios where differences in character encoding schemes can be exploited for bypassing filters in web applications that try for example to eliminate Cross-Site Scripting attacks. However, as discussed above, most of the scenarios require conditions that are rather unlikely to be met in the real world.

4 Conclusion

As a result, one can say that ambiguities between the different encodings are limited because of they all feature compatibility to ASCII to a certain degree. Least problematic with respect to web applications are the differences between the single-byte or variable width encodings like the ISO 8859-x family, UTF-8, or Big5 as they are supersets of ASCII. Only few of them incorporate small changes within the range of ASCII characters.

For web applications and content filters that aim for example at Cross-Site Scripting attacks, the following points are valid advises:

- The filter must work with the same character encoding that the web application delivers its content in. This is especially important if the web application delivers its content as UTF-16/32.
- The web application must explicitly tell the web browser what character encoding is used for the delivered content. The preferred way to do this is by using the HTTP content type header directly. This eliminates the need for algorithms that try to automatically detect the encoding of the delivered page in the browser.
- If the used character encoding is UTF-8, the filter should be aware of overlong UTF-8 sequences and should not allow them to pass the filter. Although most current cli-

ents should treat them properly, this measure reduces the risk for those applications that still decode and interpret these sequences.

So, for a filter trying to remove malicious code fragments there are relatively few problems with respect to the different character encodings. The area where much more problems exist is when it comes to the question of what characters, bytes or sequences of them should be filtered, because parsers do not always strictly adhere to the specification.

For example the Microsoft Internet Explorer ignores null bytes in HTML files altogether. This means that an attacker can intersperse arbitrary numbers of null bytes in for example a script tag. Internet Explorer will ignore them and interpret the malformed script tag as script tag. Filters that do not know about such parser-specific idiosyncrasies will only perform strict searches for the character sequences of a script tag. Another example is that Internet Explorer interprets the grave accent, also known as backtick or backquote, as a single quote.

So, filters must know about all standard deviations of all web browsers or other applications that consume the delivered content. As this is virtually impossible any filtering on the server side can only help to a certain degree even if the effects of different character encodings are solved.

5 References

- [1] F. Yergeau: *UTF-8, a standard transformation format of ISO 10646*, RFC 3629, November 2003, <http://www.ietf.org/rfc/rfc3629.txt>
- [2] Microsoft Security Bulletin MS00-078: *Web Server Folder Traversal Vulnerability*, October 2000, <http://www.microsoft.com/technet/security/bulletin/MS00-078.msp>
- [3] Securityfocus Vulnerability Database: *Microsoft IIS and PWS Extended Unicode Directory Traversal Vulnerability*, Bugtraq ID 1806, October 2000, <http://www.securityfocus.com/bid/1806/>
- [4] Unicode Consortium: *FAQ – UTF-8, UTF-16, UTF-32 & BOM*, http://www.unicode.org/faq/utf_bom.html
- [5] World Wide Web Consortium: *HTML 4.01 Specification*, December 1999, <http://www.w3.org/TR/1999/REC-html401-19991224/>
- [6] Microsoft Global Development and Computing Portal: *Windows Codepage 932*, May 2005, <http://www.microsoft.com/globaldev/reference/dbcs/932.msp>