Technical Reports

L2/14-181

Proposed Update Unicode Technical Report #36

UNICODE SECURITY CONSIDERATIONS

Editors	Mark Davis (markdavis@google.com), Michel Suignard (michel@suignard.com)
Date	2014-07-29
This Version	http://www.unicode.org/reports/tr36/tr36-14.html
Previous Version	http://www.unicode.org/reports/tr36/tr36-13.html
Latest Version	http://www.unicode.org/reports/tr36/
Latest Proposed Update	http://www.unicode.org/reports/tr36/proposed.html
Revision	14

Summary

Because Unicode contains such a large number of characters and incorporates the varied writing systems of the world, incorrect usage can expose programs or systems to possible security attacks. This is especially important as more and more products are internationalized. This document describes some of the security considerations that programmers, system analysts, standards developers, and users should take into account, and provides specific recommendations to reduce the risk of problems.

Status

This is a ***draft*** *document which may be updated, replaced, or superseded by other documents at any time. Publication does not imply endorsement by the Unicode Consortium. This is not a stable document; it is inappropriate to cite this document as other than a work in progress.*

> ***A Unicode Technical Report (UTR)*** *contains informative material. Conformance to the Unicode Standard does not imply conformance to any UTR. Other specifications, however, are free to make normative references to a UTR.*

Please submit corrigenda and other comments with the online reporting form

[Feedback]. Related information that is useful in understanding this document is found in the References. For the latest version of the Unicode Standard see [Unicode]. For a list of current Unicode Technical Reports see [Reports]. For more information about versions of the Unicode Standard, see [Versions].

Contents

1 Introduction

The Unicode Standard represents a very significant advance over all previous methods of encoding characters. For the first time, all of the world's characters can be represented in a uniform manner, making it feasible for the vast majority of programs to be *globalized:* built to handle any language in the world.

In many ways, the use of Unicode makes programs much more robust and secure. When systems used a hodge-podge of different charsets for representing characters, there were security and corruption problems that resulted from differences between those charsets, or from the way in which programs converted to and from them.

However, because Unicode contains such a large number of characters, and incorporates the varied writing systems of the world, incorrect usage can expose programs or systems to possible security attacks. This document describes some of the security considerations that programmers, system analysts, standards developers, and users should take into account.

For example, consider visual spoofing, where a similarity in visual appearance fools a user and causes him or her to take unsafe actions.

> Suppose that the user gets an email notification about an apparent problem in their Citibank account. Security-savvy users realize that it might be a spoof; the HTML email might be presenting the URL http://citibank.com/... visually, but might be hiding the *real* URL. They realize that even what shows up in the status bar might be a lie, because clever Javascript or ActiveX can work around that. (And users may are likely to have these turned on, unless they know to turn them off.) They click on the link, and carefully examine the browser's address box to make sure that it is actually going to http://сitibank.com/.... They see that it is, and use their password. However, what they saw was wrong—it is actually going to a spoof site with a fake "сitibank.com", using the Cyrillic letter that looks precisely like a 'c'. They use the site without suspecting, and the password ends up compromised.

This problem is not new to Unicode: it was possible to spoof even with ASCII characters alone. For example, "inteI.com" uses a capital I instead of an L. The infamous example here involves "paypaI.com":

> ... Not only was "Paypai.com" very convincing, but the scam artist even goes one step further. He or she is apparently emailing PayPal customers, saying they have a large payment waiting for them in their account.
>
> The message then offers up a link, urging the recipient to claim the funds. However, the URL that is displayed for the unwitting victim uses a capital "i" (I), which looks just like a lowercase "L" (l), in many computer fonts. ...

(for details, see the Unicode Security FAQ)

While some browsers prevent this spoof by lowercasing domain names, others do not.

Thus to a certain extent, the new forms of visual spoofing available with Unicode are a matter of degree and not kind. However, because of the very large number of Unicode characters (over 107,000 in the current version), the number of opportunities for visual spoofing is significantly larger than with a restricted character set such as ASCII.

1.1 Structure

This document is organized into two sections: visual security issues and non-visual security issues. Each section presents background information on the kinds of problems that can occur, and lists specific recommendations for reducing the risk of such problems. For background information, see the References and the Unicode FAQ on *Security Issues* [FAQSec].

A URL is technically a type of uniform resource identifier (URI), but in many technical documents and verbal discussions, URL is often used as a synonym for URI or IRI, and this is not considered a problem. That practice is followed here.

2 Visual Security Issues

Visual spoofs depend on the use of *visually confusable* strings: two different strings of Unicode characters whose appearance in common fonts in small sizes at typical screen resolutions is sufficiently close that people easily mistake one for the other.

There are no hard-and-fast rules for visual confusability: many characters look like others when used with sufficiently small sizes. "Small sizes at screen resolutions" means fonts whose ascent plus descent is from 9 to 12 pixels for most scripts, and somewhat larger for scripts, such as Japanese, where the users typically have larger sizes. Confusability also depends on the style of the font: with a traditional Hebrew style, many characters are only distinguishable by fine differences which may be lost at small sizes. In some cases sequences of characters can be used to spoof: for example, "rn" ("r" followed by "n") is visually confusable with "m" in many sans-serif fonts.

Where two different strings can always be represented by the same sequence of glyphs, those strings are called *homographs*. For example, "AB" in Latin and "AB" in Greek are homographs. Spoofing is not dependent on just homographs; if the visual appearance is close enough at small sizes or in the most common fonts, that can be sufficient to cause problems. Some people use the term *homograph* broadly, encompassing all visually confusable strings.

Two characters with similar or identical glyph shapes are not visually confusable if the positioning of the respective shapes is sufficiently different. For example, foo·com (using the hyphenation point instead of the period) should be distinguishable from foo.com by the positioning of the dot.

It is important to be aware that identifiers are special-purpose strings used for identification, strings that are deliberately limited to particular repertoires for that purpose. Exclusion of characters from identifiers does not affect the general use of those characters, such as within documents.

The remainder of this section is concerned with identifiers that can be confused by ordinary users at typical sizes and screen resolutions. For examples of visually confusable characters, see *Section 4, Confusable Detection* in *UTS #39: Unicode Security Mechanisms* [UTS39].

It is also important to recognize that the use of visually confusable characters in spoofing is often overstated. Moreover, confusable characters account for a small proportion of phishing problems: most are cases like "secure-wellsfargo.com". For more information, see the Unicode Security FAQ.

2.1 Internationalized Domain Names

Visual spoofing is an especially important subject given the introduction in 2003 of Internationalized Domain Names (IDN) [IDNA2003]. There is a natural desire for people to see domain names in their own languages and writing systems; English speakers can understand this if they consider what it would be like if they always had to type Web addresses with Japanese characters. IDNs represent a very significant advance for most people in the world. However, the larger repertoire of characters results in more opportunities for spoofing. Proper implementation in browsers and other programs is required to minimize security risks while still allowing for effective use of non-ASCII characters.

Internationalized Domain Names are, of course, not the only cases where visual spoofing can occur. One example is a message offering to install software from "IBM", authenticated with a certificate in which the "M" character happens to be the Russian (Cyrillic) character that looks precisely like the English "M". Wherever strings are used as identifiers, this kind of spoofing is possible.

IDNs provide a good starting point for a discussion of visual spoofing, and are the focus of the next part of this section. In 2010, there was a update to [IDNA2003] called [IDNA2008]. Because the concepts and recommendations discussed here can be generalized to the use of other types of identifiers, both [IDNA2003] and [IDNA2008] will be used in examples. For background information on identifiers, see UAX #31: *Identifier and Pattern Syntax* [UAX31]. For more information on how to handle international domain names in a compatible fashion, see *UTS #46: Unicode IDNA Compatibility Processing* [UTS46].

Fortunately the design of IDN prevents a huge number of spoofing attacks. All conformant users of [IDNA2003] are required to process domain names to convert what are called *compatibility-equivalent* characters into a unique form using a process called compatibility normalization (NFKC)—for more information on this, see [UAX15]. This processing eliminates most possibilities for visual spoofing by mapping away a large number of visually confusable characters and sequences. For example, characters like the halfwidth Japanese *katakana* character ｶ are converted to the regular character カ, and single ligature characters like "ﬁ" to the sequence of regular characters "fi". Unicode contains the "ä" (a-umlaut) character, but also contains a free-standing umlaut (" ̈") which can be used in combination with any character, including an "a". The compatibility normalization will convert any sequence of "a" plus " ̈" into the regular "ä". ([IDNA2008] disallows these compatibility characters as output, but allows them to be mapped on input.)

Thus someone cannot spoof an *a-umlaut* with *a* + *umlaut*; it simply results in the same domain name. See the example in *Table 1, Safe Domain Names* . The String column shows the actual characters; the UTF-16 column shows the underlying encoding and the Punycode column shows the internal format of the domain name. This is the result of applying the ToASCII() operation [RFC3490] to the original IDN, which is the way this IDN is stored and queried in the DNS (Domain Name System).

Table 1. Safe Domain Names

	String	UTF-16	Punycode	Comments
1a	ät.com	**0061 0308** 0074 002E 0063 006F 006D	xn--t-zfa.com	Uses the decomposed form, a plus umlaut
1b	ät.com	**00E4** 0074 002E 0063 006F 006D	xn--t-zfa.com	The decomposed form ends up being identical to the composed form, in IDNA

Similarly, for most scripts, two accents that do not interact typographically are put into a determinate order when the text is normalized. Thus the sequence <x, dot_above, dot_below> is reordered as <x, dot_below, dot_above>. This ensures that the two sequences that look identical (ẋ̣ and ẋ̣) have the same representation.

Note: The demo at [IDN-Demo] can be used to demonstrate the results of processing different domain names. That demo was also used to get the Punycode values shown in *Table 1, Safe Domain Names* .

The [IDNA2003] and [UTS46] processing also removes case distinctions by performing a *casefolding* to reduce characters to a lowercase form. This is helps avoid spoofing problems, because characters are generally more distinctive in their lowercase forms. That means that implementers can focus on just dealing with the lowercase characters. There are some cases where people will want to see certain special differences preserved in display. For more information, and information about characters allowed in IDN, see *UTS #46: Unicode IDNA Compatibility Processing* [UTS46].

> **Note**: Users expect diacritical marks to distinguish domain names. For example, the domain names "resume.com" and "résumé.com" are (and should be) distinguished. In languages where the spelling may allow certain words with and without diacritics, registrants would have to register two or more domain names to cover user expectations (just as one may register both "analyze.com" and "analyse.com" to cover variant spellings). The registry can support this automatically by using a technique known as "bundling".

Although normalization and casefolding prevent many possible spoofing attacks, visual spoofing can still occur with many IDNs. This poses the question of which parts of the infrastructure using and supporting domain names are best suited to minimize possible spoofing attacks.

Some of the problems of visual spoofing can be best handled on the registry side, while others can be best handled on the side of the *user agent*: browsers, emailers, and other programs that display and process URLs. The registry has the most data available about alternative registered names, and can process that information the most efficiently at the time of registration, using policies to reduce visual spoofing. For example, given the method described in *Section 4, Confusable Detection* in *UTS #39: Unicode Security Mechanisms* [UTS39], the registry can easily determine if a proposed registration could be visually confused with an existing one; that determination is much more difficult for user agents because of the sheer number of combinations that they would have to check.

However, there are certain issues much more easily addressed by the user agent:

- the user agent has more control over the display of characters, which is crucial to spoofing
- there are legitimate cases of visually confusable characters that one may want to allow *after* alerting the user, such as single-script confusables discussed below
- one cannot depend on all registries being responsive to security issues
- due to the decentralized nature of DNS, a registry for a domain does not control subdomains: thus the registry for a top-level domain (TLD) like ".com" may not control the labels accepted by a subdomain like "blogspot.com".

Thus the problem of visual spoofing is most effectively addressed by a combination of strategies involving user agents and registries.

2.2 Mixed-Script Spoofing

Visually confusable characters are not usually unified across scripts. Thus a Greek *omicron* is encoded as a different character from the Latin "o", even though it is usually identical or nearly identical in appearance. There are good reasons for this: often the characters were separate in legacy encodings, and preservation of those distinctions was necessary for data to be converted to Unicode and back without loss. Moreover, the characters generally have very different behavior: two visually confusable characters may be different in casing behavior, in category (letter versus number), or in numeric value. After all, ASCII does not unify lowercase letter l and digit 1, even though those are visually confusable. (Many fonts always distinguish them, but many others do not.) Encoding the Cyrillic character б (corresponding to the letter "b") by using the numeral 6, would clearly have been a mistake, even though they are visually confusable.

However, the existence of visually confusable characters across scripts offers numerous opportunities for spoofing. For example, a domain name can be spoofed by using a Greek omicron instead of an 'o', as in example 1a in *Table 2, Mixed-Script Spoofing*.

Table 2. Mixed-Script Spoofing

	String	UTF-16	Punycode	Comments
1a	top.com	0074 **03BF** 0070 002E 0063 006F 006D	xn--tp-jbc.com	Uses a Greek omicron in place of the o

1b	top.com	0074 **006F** 0070 002E 0063 006F 006D	top.com	

There are many legitimate uses of mixed scripts. For example, it is quite common to mix English words (with Latin characters) in other languages, including languages using non-Latin scripts. For example, one could have XML-документы.com (which would be a site for "XML documents" in Russian). Even in English, legitimate product or organization names may contain non-Latin characters, such as Ωmega, Teχ, Toys-Я-Us, or HλLF-LIFE. The lack of IDNs in the past has also led to the usage in some registries (such as the .ru top-level domain) where Latin characters have been used to create pseudo-Cyrillic names in the .ru (Russian) top-level domain. For example, see http://caxap.ru/ (caxap means sugar in Russian).

For information on detecting mixed scripts, see *Section 5, Mixed Script Detection* of *UTS #39: Unicode Security Mechanisms [UTS39].*

Cyrillic, Latin, and Greek represent special challenges, because the number of common glyphs shared between them is so high, as can be seen from *Section 4, Confusable Detection* in *UTS #39: Unicode Security Mechanisms* [UTS39]. It may be possible to compose an entire domain name (except the top-level domain) in Cyrillic using letters that will be essentially always identical in form to Latin letters, such as "scope.com": with "scope" in Cyrillic looking just like "scope" in Latin. Such spoofs are called *whole-script spoofs,* and the strings that cause the problem are correspondingly called *whole-script confusables.*

2.3 Single-Script Spoofing

Spoofing with characters entirely within one script, or using characters that are common across scripts (such as numbers), is called *single-script spoofing*, and the strings that cause it are correspondingly called *single-script confusables*. While compatibility normalization and mixed-script detection can handle the majority of spoofing cases, they do not handle single-script confusables. Especially at the smaller font sizes in the context of an address bar, any visual confusables within a single script can be used in spoofing. Importantly, these problems can be illustrated with common, widely available fonts on widely available operating systems—the problems are not specific to any single vendor.

Consider the examples in *Table 3, Single-Script Spoofing*, all in the same script. In each numbered case, the strings will look identical or nearly identical in most browsers.

Table 3. Single-Script Spoofing

	String	UTF-16	Punycode	Comments
1a	a-b.com	0061 **2010** 0062 002E 0063 006F 006D	xn--ab-v1t.com	Uses a real hyphen, instead of the ASCII hyphen-minus
1b	a-b.com	0061 **002D** 0062 002E 0063 006F 006D	a-b.com	

2a	so̷s.com	0073 **006F 0337** 0073 002E 0063 006F 006D	xn--sos-rjc.com	Uses o + combining slash
2b	søs.com	0073 **00F8** 0073 002E 0063 006F 006D	xn--ss-lka.com	
3a	z̵o.com	**007A 0335** 006F 002E 0063 006F 006D	xn--zo-pyb.com	Uses z + combining bar
3b	ƶo.com	**01B6** 006F 002E 0063 006F 006D	xn--o-zra.com	
4a	an͂o.com	0061 **006E 0342** 006F 002E 0063 006F 006D	xn--ano-0kc.com	Uses n + greek perispomeni
4b	año.com	0061 **00F1** 006F 002E 0063 006F 006D	xn--ao-zja.com	
5a	ʣe.org	**02A3** 0065 002E 006F 0072 0067	xn--e-j5a.org	Uses d-z digraph
5b	dze.org	**0064 007A** 0065 002E 006F 0072 0067	dze.org	

Examples exist in various scripts. For instance, 'rn' was already mentioned above, and the sequence अ + ा typically looks identical to आ.

In most cases two sequences of accents that have the same visual appearance are put into a canonical order. This does not happen, however, for certain scripts used in Southeast Asia, so reordering characters may be used for spoofs in those cases. See *Table 4, Combining Mark Order Spoofing*.

Table 4. Combining Mark Order Spoofing

	String	UTF-16	Punycode	Comments
1a	လို.com	101C **102D** 102F	xn--gjd8ag.com	Reorders two combining marks
1b	လုိ.com	101C 102F **102D**	xn--gjd8af.com	

2.4 Inadequate Rendering Support

An additional problem arises when a font or rendering engine has inadequate support for characters or sequences of characters that should be visually distinguishable, but do not appear that way. In *Table 5, Inadequate Rendering Support* , examples 1a and 1b show the cases of lowercase L and digit one, mentioned above. While this depends on the font, on the computer used to write this document, roughly 30% of the fonts display glyphs that are essentially identical. In example 2a, the *a-umlaut* is followed by another *umlaut*. The Unicode Standard guidelines indicate that the second *umlaut* should be 'stacked' above the first, producing a distinct visual difference. However, as example 2a shows, common fonts will simply superimpose the second *umlaut*; and if the positioning is close enough, the user will not see a difference between 2a and 2b. Examples 3 a, b, and c show an even worse case. The *underdot* character in 3a should appear under the 'l', but as rendered with many fonts, it appears under the 'e'. It is thus visually confusable with 3b (where the *underdot* is under the e) or the equivalent normalized form 3c.

Table 5. Inadequate Rendering Support

	String	UTF-16	Punycode	Comments
1a	al.com	0061 **006C** 002E 0063 006F 006D	al.com	1 and l may appear alike, depending on font.
1b	a1.com	0061 **0031** 002E 0063 006F 006D	a1.com	
2a	ä̈t.com	**00E4 0308** 0074 002E 0063 006F 006D	xn--t-zfa85n.com	a-umlaut + umlaut
2b	ät.com	**00E4** 0074 002E 0063 006F 006D	xn--t-zfa.com	
3a	eḷ.com	**0065** 006C **0323** 002E 0063 006F 006D	xn--e-zom.com	Has a dot under the l; may appear under the e
3b	ẹl.com	**0065 0323** 006C 002E 0063 006F 006D	xn--l-ewm.com	
3c	ẹl.com	**1EB9** 006C 002E 0063 006F 006D	xn--l-ewm.com	

Certain Unicode characters are invisible, although they may affect the rendering of the characters around them. An example is the *joiner* character, used to request a cursive connection such as in Arabic. Such characters may often be in positions where they have no visual distinction, and are thus discouraged for use in identifiers except in specific contexts. For more information, see *UTS #46: Unicode IDNA Compatibility Processing* [UTS46].

A sequence of ideographic description characters may be displayed as if it were a CJK character; thus they are also discouraged.

2.4.1 Malicious Rendering

Font technologies such as TrueType/OpenType are extremely powerful. A glyph in such a font actually may use a small programs to transform the shape radically according to resolution, platform, or language. This is used to chose an optimal shape for the character under different conditions. However, it can also be used in a security attack, because it is powerful enough to change the appearance of, say "$**1**00.00" on the screen to "$**2**00.00" when printed.

In addition Cascading Style Sheets (CSS) can change to a different font for printing versus screen display, which can open up the use of more confusable fonts.

These problems are not specific to Unicode. To reduce the risk of this kind of exploit, programmers and users should only allow trusted fonts in such circumstances.

2.5 Bidirectional Text Spoofing

Some characters, such as those used in the Arabic and Hebrew script, have an inherent right-to-left writing direction. When these characters are mixed with characters of other scripts or symbol sets which are displayed left-to-right, the resulting text is called bidirectional (abbreviated as *bidi*). The relationship between the memory representation of the text (logical order) and the display appearance (visual order) of bidi text is governed by *UAX #9: Unicode Bidirectional Algorithm* [UAX9].

Because some characters have weak or neutral directionalities, as opposed to strong left-to-right or right-to-left, the Unicode Bidirectional Algorithm uses a precise set of rules to determine the final visual rendering. However, presented with arbitrary sequences of text, this may lead to text sequences which may be impossible to read intelligibly, or which may be visually confusable. To mitigate these issues, the [IDNA2003] specification requires that:

- each label of a host name must not use both right-to-left and left-to-right characters,
- a label using right-to-left character must start and end with right-to-left characters.

The [IDNA2008] specification improves these rules, allowing some sequences that are incorrectly forbidden by the above rules, and disallowing others that can cause visual confusion.

In addition, the IRI specification [RFC3987] extends those requirements to other components of an URL, not just the host name labels. Not respecting them would result in insurmountable visual confusion. A large part of the confusability in reading an URL containing bidi characters is created by the weak or neutral directionality property of many URL delimiters such as '/', '.', '?' which makes them change directionality depending on their surrounding characters. This is shown with the dots in *Table 6, Bidi Examples* , where they are colored the same as the preceding label. Notice that the placement of that following punctuation may vary.

Table 6. Bidi Examples

	Samples
1	http://سلام.دائم.com
2	http://سلام.a.دائم.com

Adding the left-to-right label "a" between the two Arabic labels splits them up and reverses their display order, as seen in example #2 in *Table 6, Bidi Examples*. The IRI specification [RFC3987] provides more examples of valid and invalid IRIs using various mixes of bidi text.

To minimize the opportunities for confusion, it is imperative that the [IDNA2008] and IRI requirements concerning bidi processing be fully implemented in the processing of host names containing bidi characters. Nevertheless, even when these requirements are met, reading IRIs correctly is not trivial. Because of this, mixing right-to-left and left-to-right characters should be done with great care when creating bidi IRIs.

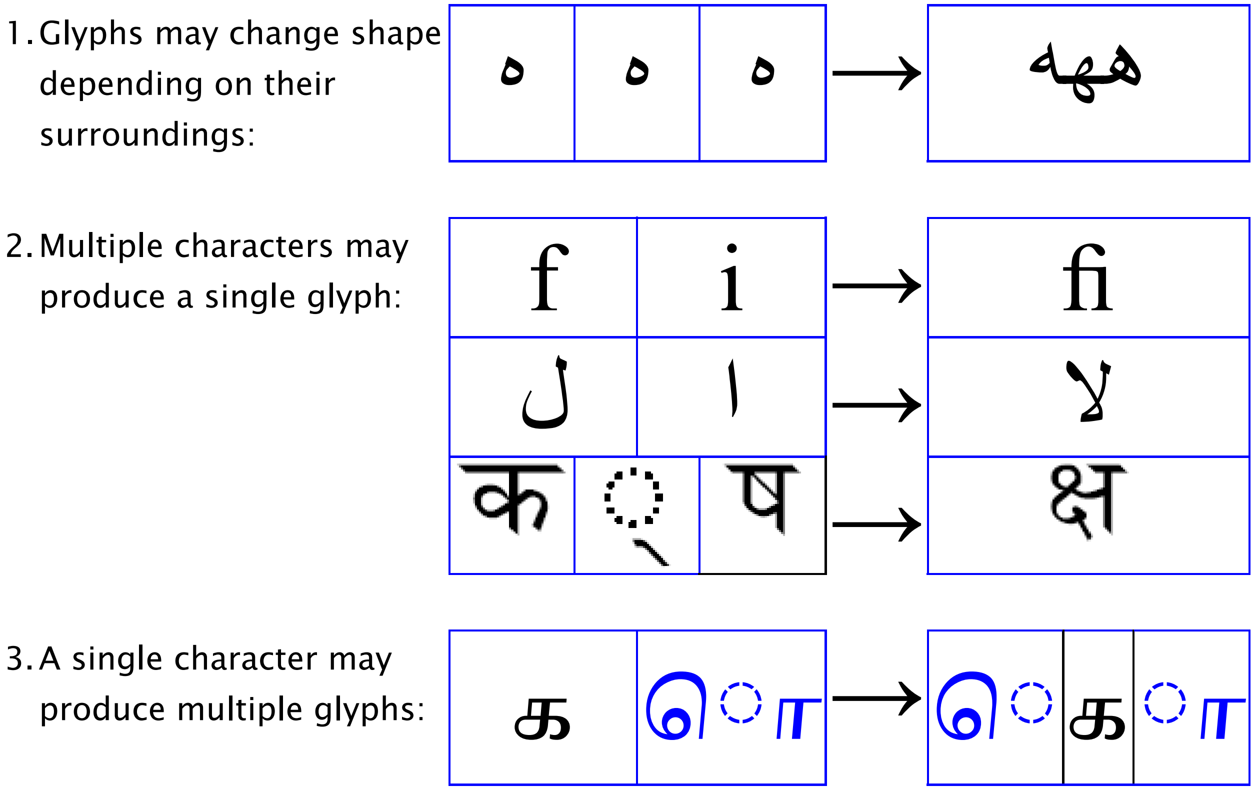
Recommendations:

- Never allow bidi override characters.
- As much as possible, avoid mixing right-to-left and left-to-right characters in a single name.
- When right-to-left characters are used, limit the usage of left-to-right characters to well-known cases such as TLD names and URL scheme names (such as http, ftp, mailto, and so on).
- Minimize the use of digits in host names and other components of IRIs containing right-to-left characters.
- Keep IRIs containing bidi content simple to read.
- Use reverse-bidi (visual order -> storage order) to detect possible bidi spoofs. That is, one can apply bidi, then reverse bidi: if the result does not match the original storage order, then the visual reading is ambiguous and the string can be rejected. This is, however, subject to false positives, so this should probably be presented to users for confirmation.

2.5.1 Glyphs in Complex Scripts

In complex scripts such as Arabic and South Asian scripts, characters may change shape according to the surrounding characters, as shown in *Table 7, Glyphs in Complex Scripts*. Note that this also occurs in higher-end typography in English, as illustrated by the "fi" ligature. Two characters might be visually distinct in a stand-alone form, but not be distinct in a particular context.

Table 7. Glyphs in Complex Scripts


1. Glyphs may change shape depending on their surroundings:
ه ه ه →ههه
2. Multiple characters may produce a single glyph:
f i → ﬁ
ل ا → لا
क ् ष → क्ष
3. A single character may produce multiple glyphs:
க ொ → ெ க ா

Some complex scripts are encoded with a so-called *font-encoding,* where non-private-use characters are misused as other characters or parts of characters. These present special risks, because the encodings are not identified, and the visual interpretation of the characters depends entirely on the font, and is completely disconnected from the underlying characters. Luckily such font-encodings are seldom used, and their use is decreasing rapidly with the growth of Unicode.

2.6 Syntax Spoofing

Spoofing syntax characters can be even worse than regular characters, as illustrated in *Table 8, Syntax Spoofing*. For example, U+2044 (⁄) FRACTION SLASH can look like a regular ASCII '/' in many fonts—ideally the spacing and angle are sufficiently different to distinguish these characters. However, this is not always the case. When this character is allowed, the URL in line 1 may appear to be in the domain **macchiato.com**, but is actually in a particular subzone of the domain **bad.com**.

Table 8. Syntax Spoofing

	URL	Subzone	Domain
1	http://macchiato.com⁄x.bad.com	macchiato.com⁄x	bad.com
2	http://macchiato.com?x.bad.com	macchiato.com?x	bad.com
3	http://macchiato.com.x.bad.com	macchiato.com.x	bad.com
4	http://macchiato.com#x.bad.com	macchiato.com#x	bad.com

Where there are visual confusables other syntax characters can be similarly spoofed, as in lines 2 through 4. Nameprep [RFC3491] and [UTS46] disallow many such cases, such as such as U+2024 (․) ONE DOT LEADER. However, not all syntax spoofs are disallowed.

Of course, these types of spoofs do not require IDNs. For example, in the following the real domain name, **bad.com**, is also obscured for the casual user, who may not realize that "--" does not terminate the domain name.

http://macchiato.com--long-and-obscure-list-of-characters.bad.com?findid=12

In retrospect, it would have been much better if domain names were customarily written with the most significant label first. The following hypothetical display would be harder to spoof: it is easy to see that the top level is "com.bad".

http://**com.bad**.org/x.example?findid=12
http://**com.bad**.org--long-and-obscure-list-of-characters.example?findid=12

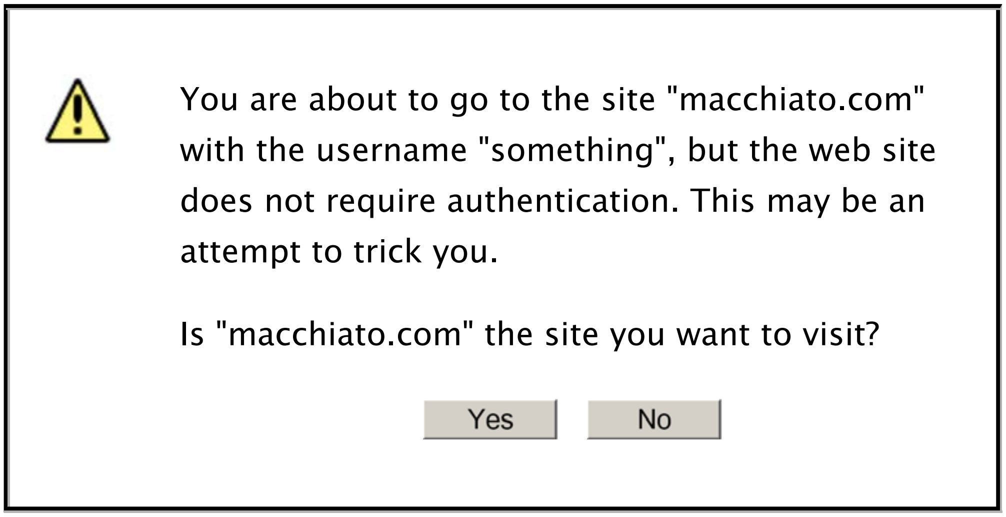
However, that would be an impossible change at this point. However, much the same effect can be produced by always visually distinguishing the domain, for example:

http://**macchiato.com**
http://**bad.com**
http://macchiato.com/**x.bad.com**

http://**macchiato.com--long-and-obscure-list-of-characters.bad.com**?findid=12
http://**220.135.25.171**/amazon/index.html

Such visual distinction could be in different ways, such as highlighting in an address box as above, or extracting and displaying the domain name in a noticeable place.

User agents already have to deal with syntax issues. For example, Firefox gives something like the following alert when given the URL http://something@macchiato.com:



You are about to go to the site "macchiato.com" with the username "something", but the web site does not require authentication. This may be an attempt to trick you.

Is "macchiato.com" the site you want to visit?

Such a mechanism can be used to alert the user to cases of syntax spoofing.

2.6.1 Missing Glyphs

It is very important not to show a missing glyph or character with a simple "?", because every such character is visually confusable with a real question mark. Instead, follow the Unicode guidelines for displaying missing glyphs using a rounded-rectangle, as listed in *Appendix A* *Script Icons* and described in *Section 5.3, Unknown and Missing Characters* of [Unicode].

Private use characters must be avoided in identifiers, except in closed environments. There is no predicting what either the visual display or the programmatic interpretation will be on any given machine, so this can obviously lead to security problems. This is not a problem for IDNs, because private use characters are excluded in all specifications: [IDNA2003], [IDNA2008], and [UTS46].

What is true for private use characters is doubly true of unassigned code points. Secure systems will not use them: any future Unicode Standard could assign those codepoints to any new character. This is especially important in the case of certification.

2.7 Numeric Spoofs

Turning away from the focus on domain names for a moment, there is another area where visual spoofs can be used. Many scripts have sets of decimal digits that are different in shape from the typical European digits. For example, Bengali has {০ ১ ২ ৩ ৪ ৫ ৬ ৭ ৮ ৯}, while Oriya has {୦ ୧ ୨ ୩ ୪ ୫ ୬ ୭ ୮ ୯}. Individual digits may have the same shapes as digits from other scripts, even digits of different values. For example, the Bengali string "৪২" is visually confusable with the European digits "**89"**, but actually has the numeric value 42! If software interprets the numeric value of a string of digits without detecting that the digits are from different or inappropriate scripts, such spoofs can be used.

2.8 IDNA Ambiguity

IDNA2008, just approved in 2010, opens up new opportunities for spoofing. In the 2003 version of international domain names, a correctly processed URL containing Unicode characters always resolved to the same Punycode URL for lookup. IDNA2008, in certain cases, will resolve to a different Punycode URL. Thus the same URL, whether typed in by the user or present in data (such as in an href) will resolve to two different locations, depending on whether the user is using a browser on the pre-2010 international domain name specification or the post-2010 specification. For more information on this topic, see *UTS #46: Unicode IDNA Compatibility Processing* [UTS46] and [IDN_FAQ].

2.8.1 Punycode Spoofs

The Punycode transformation is relatively dense. That means that it is fairly likely that arbitrary words after the "xn--" will result in valid labels. For example, see *Table 8a. Punycode Spoofing*.

Table 8a. Punycode Spoofing

	URL	Punycode URL
1	http://䕮䕵䚞䤏.com	http://xn--google.com
2	http://䁾.com	http://xn--cnn.com
3	http://岍岊岊岅岉岎.com	http://xn--citibank.com

These examples demonstrate that the common tactic of displaying Punycode for suspicious URLs or for URLs with languages or scripts not in the user's settings can actually backfire, producing display results that are *more* likely to mislead the user. For example, if a user is unfamiliar with Chinese but knows Latin characters, she is more likely to be mislead by the Punycode URL “http://xn--cnn.com” than by the corresponding Unicode URL “http://䁾.com”. More examples can be created with the demo at [IDN-Demo].

2.9 Techniques

This section lists techniques for reducing the risks of visual spoofing. These techniques are referenced by *Section 2.10, Recommendations.*

2.9.1 Casefolded Format

Many opportunities for spoofing can be removed by using a *casefolded* format. This format, defined by the Unicode Standard, produces a string that only contains lowercase characters where possible.

However, four characters that require special handling in casefolding, where the pure casefolded format of a string as defined by the Unicode Standard is not desired. For example, the character U+03A3 "Σ" *capital sigma* lowercases to U+03C3 "σ" *small sigma* if it is followed by another letter, but lowercases to U+03C2 "ς" *small final sigma* if it is not. Because both σ and ς have a case-insensitive match to Σ, and the casefolding algorithm needs to map both of them together (so that transitivity is maintained), only one of them appears in the casefolded form.

> When σ comes after a cased letter, and not before a cased letter (where certain ignorable characters can come in between), it should be transformed into ς. For more details, see the test for Final_Sigma as provided in Table 3-15 of [Unicode].

For more information, see *UTS #46: Unicode IDNA Compatibility Processing* [UTS46]. For more information on case mapping and folding, see the following: *Section 3.13, Default Case Operations*, *Section 4.2; Case Normative*; and *Section 5.18, Case Mappings* of [Unicode].

2.9.2 Mapping and Prohibition

Mapping and prohibition are two useful techniques to reduce the risk of spoofing that can be applied to identifiers. A number of characters are included in Unicode for compatibility. *Compatibility Normalization* (NFKC) can be used to map these characters to the regular variants. For example, a halfwidth Japanese *katakana* character ｶ is mapped to the regular character カ. Additional mappings can be added beyond compatibility mappings, for example, [IDNA2003] adds the following:

`200D; ZERO WIDTH JOINER` maps to nothing (that is, is removed)
`0041; 0061;` Case maps 'A' to 'a'
`20A8; 0072 0073;` Additional folding, mapping Rs to "rs"

In addition, characters may be prohibited. For example, IDNA2003 prohibits *space* and *no-break space* (U+00A0). Instead of removing a ZERO WIDTH JOINER, or mapping Rs to "rs", one could prohibit these characters. There are pluses and minuses to both approaches. If compatibility characters are widely used in practice in entering text, it is much more user-friendly to remap them. This also extends to deletion; for example, the ZERO WIDTH JOINER is commonly used to affect the presentation of characters in languages such as Hindi or Arabic. In this case, text copied into the address box may often contain the character.

Where this is not the case, however, it may be advisable to simply prohibit the character. It is unlikely, for example, that ㋕ would be typed by a Japanese user, nor that it would need to work in copied text.

Where both mapping and prohibition are used, the mapping should be done before the prohibition, to ensure that characters do not "sneak past". For example, the Greek character TONOS (΄) ends up being prohibited in [IDNA2003] , because it normalizes to *space* + *acute*, and *space* itself is prohibited.

Many languages have words whose correct spelling requires the use of certain invisible characters, especially the Join_Control characters:

200C ZERO WIDTH NON-JOINER
200D ZERO WIDTH JOINER

For that reason, as of Version 5.1 of the Unicode Standard the recommendations for identifiers were modified to allow these characters in certain circumstances. (For more information, see *UAX #31: Unicode Identifier and Pattern Syntax* [UAX31].) There are very stringent constraints on the use of these characters, so that they are only allowed with certain scripts, and in certain circumscribed contexts. In particular, in Indic scripts the ZWJ and ZWNJ may only be used in combination with a *virama* character. This approach is adopted in [IDNA2008] and [UTS46].

Even when the join controls are constrained to being next to a *virama*, in some contexts they may not result in a different visual appearance. For example, in roughly half of the possible pairs of Malayalam consonants linked by a *virama*, the ZWNJ makes a visual difference; in the remaining cases, the appearance is the same as if only the virama were present, without a ZWNJ. Implementations or standards may thus place further restrictions on invisible characters. For join controls in Indic scripts, such restrictions would typically consist of providing a table per script, containing pairs of consonants which allow intervening *joiners*.

The Unicode property [NFKC_Casefold] can be used to get a combined casefolding, normalization, and removal of default-ignorable code points. It is the basis for the mapping of international domain names in *UTS #46: Unicode IDNA Compatibility Processing* [UTS46]. For more information, also see *UTS #39: Unicode Security Mechanisms* [UTS39].

2.10 Restriction Levels and Alerts

To help avoid problems with mixtures of scripts, *UTS #39: Unicode Security Mechanisms* [UTS39] defines *Restriction Levels*. An appropriate alert should be generated if an identifier fails to satisfy the Restriction Level chosen by the user or set in the browser. Depending on the circumstances and the level difference, the form of such alerts could be minimal, such as special coloring or icons (perhaps with a tool-tip for more information); or more obvious, such as an alert dialog describing the issue and requiring user confirmation before continuing; or even more stringent, such as disallowing the use of the identifier. Where icons are used to indicate the presence of characters from scripts, the glyphs in *Appendix A Script Icons* can be used.

The UI for giving users choice among restriction levels may vary considerably. In the case of domain names, only the middle three levels are interesting. Level 1 turns IDNs completely off, while Level 5 is not recommended for IDNs.

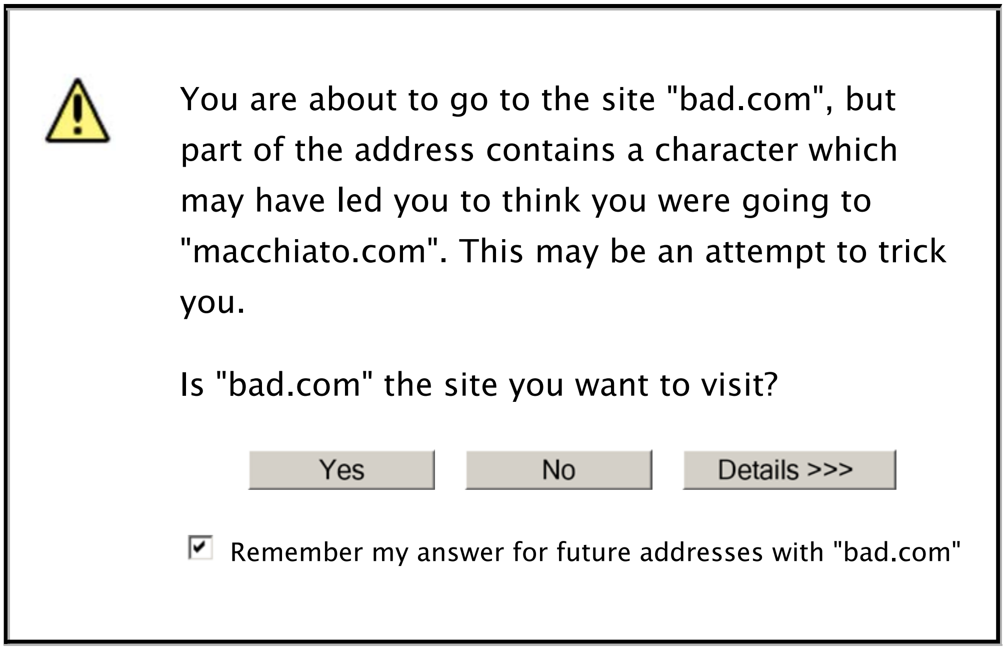
Note that the examples in Level 4 are chosen for their familiarity to English speakers. For most languages that customarily use the Latin script, there is probably little need to mix in other scripts. That is not necessarily the case for languages that customarily use a non-Latin script. Because of the widespread commercial use of English and other Latin-based languages, it is quite common to have Latin-script characters (especially

ASCII) in text that principally consists of other scripts, such as "خدمة RSS".

Section 3, Identifier Characters in *UTS #39: Unicode Security Mechanisms* [UTS39] provides for two profiles of identifiers that could be used in Restriction Levels 1 through 4. The strict profile is recommended. If the lenient profile is used, the user should have some way to choose the strict profile.

At all Restriction Levels, an appropriate alert should be generated if the domain name contains a syntax character that might be used in a spoof, as described in *Section 2.6, Syntax Spoofing*.

For example, an alert might be presented for a syntax character spoof:

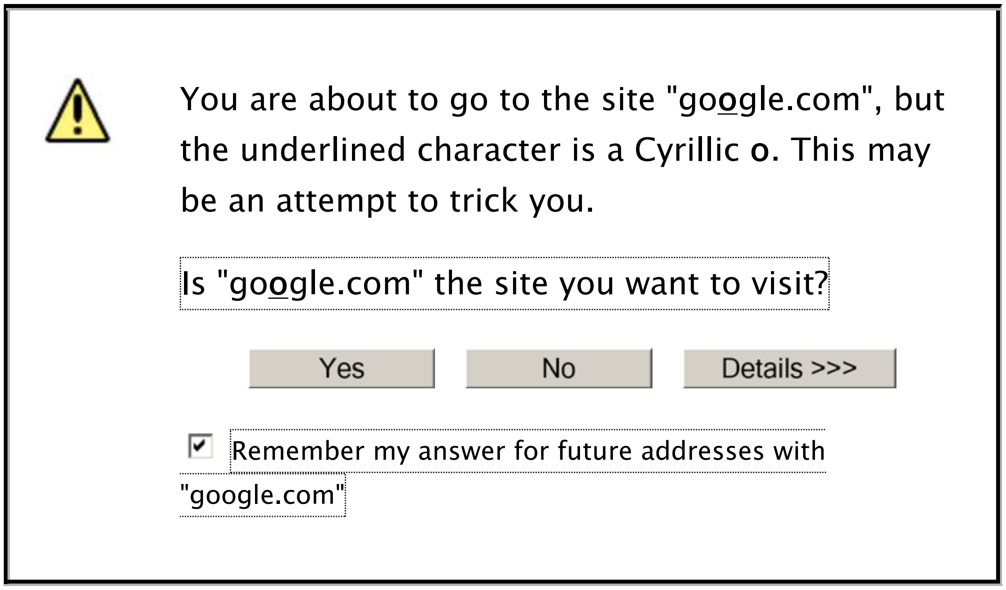


You are about to go to the site "bad.com", but part of the address contains a character which may have led you to think you were going to "macchiato.com". This may be an attempt to trick you.

Is "bad.com" the site you want to visit?

☑ Remember my answer for future addresses with "bad.com"

As another example, an alert might be presented for a mixed-script spoof:



You are about to go to the site "google.com", but the underlined character is a Cyrillic **o**. This may be an attempt to trick you.

Is "google.com" the site you want to visit?

☑ Remember my answer for future addresses with "google.com"

This alert does not need to be presented in a dialog window; there are a variety of ways to alert users, such as in an information bar.

User agents should remember when the user has accepted an alert, for say *Ωmega.com*, and permit future access without bothering the user again. This essentially builds up a whitelist of allowed values. This whitelist should contain the "nameprepped" form of each string. When used for visually confusable detection, each element in the whitelist should also have an associated transformed string as described in *Section 4, Confusable Detection* [UTS39]. If a system allows uppercase and lowercase forms, then both transforms should be available. The program should allow access to editing this whitelist directly, in case the user wants to correct the values. The whitelist may also include items known by the user agent to be 'safe'.

2.10.1 Backward Compatibility

The set of characters in the identifier profile and the results of the confusable mappings may be refined over time, so implementations should recognize and allow for that. Characters suitable for identifiers are periodically added to the Unicode Standard, and thus the data for *Section 4, Confusable Detection* [UTS39] is also periodically updated.

There may also be cases where characters are no longer recommended for inclusion in identifiers as more information becomes available about them. Thus some characters may be removed from the identifier profile in the future. Of course, once identifiers are registered they cannot be withdrawn, but new proposed identifiers that contain such characters can be denied.

2.11 Recommendations

The Unicode Consortium recommends a somewhat conservative approach at this point, because is always easier to widen restrictions than narrow them.

Some have proposed restricting domain names according to language, to prevent spoofing. In practice, that is very problematic: it is very difficult to determine the intended language of many terms, especially product or company names, which are often constructed to be neutral regarding language. Moreover, languages tend to be quite fluid; foreign words are continually being adopted. Except for registries with very special policies (such as the blocking used by some East Asian registries as described in [RFC3743]), the language association does not make too much sense. For more information, see *Appendix B, Language-Based Security*.

Instead, the Consortium recommends processing strings to remove basic equivalences, promoting adequate rendering support, and putting restrictions in place according to script, and restricting by confusable characters. While the ICANN guidelines say "top-level domain registries will [...] associate each registered internationalized domain name with one language or set of languages" [ICANN], that guidance is better interpreted as limiting to *script* rather than *language*.

Also see the security discussions in IRI [RFC3987], URI [RFC3986], and Nameprep [RFC3491].

2.11.1 Recommendations for End-Users

A. Use browsers, mail clients, and other software that have put user-agent guidelines into place to detect spoofing.

B. If registering domain names, verify that the registry follows appropriate guidelines for preventing spoofing.
C. If the desired domain name can have any whole-script or single-script confusables (such as "scope" in Latin and Cyrillic), register those as well, if "bundling" is not automatically provided by the registry.
D. Where there are alternative domain names, choose those that are less spoofable.
E. When using bidi IRIs, follow the recommendations in *Section 2.5, Bidirectional Text Spoofing* .
F. Be aware that fonts can be used in spoofing, as discussed in *Section 2.4.1, Malicious Rendering*. With documents having embedded fonts (web fonts), be aware that the content on a printed form can be different than is on the screen.

2.11.2 Recommendations for Programmers

A. When parsing numbers, detect digits of mixed scripts and unexpected scripts and alert the user.
B. When defining identifiers in programming languages, protocols, and other environments:
    1. Use the general security profile for identifiers from *Section 3, Identifier Characters* in *UTS #39: Unicode Security Mechanisms* [UTS39].
        - Note that the general security profile allows characters from *Table 3, Candidate Characters for Inclusion in Identifiers* in [UAX31], such as U+00B7 (·) MIDDLE DOT used in Catalan.
    2. For equivalence of identifiers, preprocess both strings by applying NFKC and case folding. Display all such identifiers to users in their processed form. (There may be two displays: one in the original and one in the processed form.) An example of this methodology is Nameprep [RFC3491]. Although Nameprep is currently limited to Unicode 3.2, the same methodology can be applied by implementations that need to support more up-to-date versions of Unicode.
C. In choosing or deploying fonts:
    1. If there is no available glyph for a character, *never* show a simple "?" or omit the character.
    2. Use distinctive fonts, where possible.
    3. Use a size that makes it easier to see the differences in characters. Disallow the use of font sizes that are so small as to cause even more characters to be visually confusable. Use larger sizes for East/South/South East Asian scripts, such as for Japanese and Thai.
    4. Watch for clipping, vertically and horizontally. That is, make sure that the visible area extends outside of the text width and height, to the character bounding box: the maximum extent of the shape of the glyph.
    5. Assess the font support of the OS/platform according to recommendations D1-D3 below (see also the W3C [CharMod]). If it is inadequate, work with the OS/platform vendor to address those problems, or implement special handling of problematic cases.

D. In developing rendering systems or fonts:
    1. Verify that accents do not appear to apply to the wrong characters.
    2. Follow UTN #2: *Rendering Combining Marks* in providing layout of nonspacing marks that would otherwise collide. If this is not done, follow the "Show Hidden" option of *Section 5.13, Rendering Nonspacing Marks* of [Unicode] for the display of nonspacing marks.
    3. Follow the Unicode guidelines for displaying missing glyphs using a rounded-rectangle, as described in *Section 5.3, Unknown and Missing Characters* of [Unicode]. The recommended glyphs according to scripts are shown in *Appendix A Script Icons*.

2.11.3 Recommendations for User Agents

The following recommendations are for user agents in handling domain names. The term "user agent" is interpreted broadly to mean any program that displays Internationalized Domain Names to a user, including browsers and emailers.

For information on the confusable tests mentioned below, see *Section 4, Confusable Detection* in *UTS #39: Unicode Security Mechanisms* [UTS39]. If the user can see the casefolded form, use the lowercase-only confusable mappings; otherwise use the broader mappings.

A. Follow *Section 2.10.2, Recommendations for Programmers*.

B. Display
    1. Either always show the domain name in nameprepped form [RFC3491], or make it very easy for the user to see it (see *Section 2.8.1, Casefolded Format*). For example, this could be a tooltip interface, or a separate box.
    2. Always display the domain name with a visually highlighted domain name, to prevent syntax spoofs (see *Section 2.6, Syntax Spoofing*).
    3. Always display IRIs with bidi content according to the IRI specification [RFC3987].

C. Preferences
    1. In preferences, allow the user to select the desired Restriction Level to apply to domain names. Set the default to Restriction Level 2.
    2. In preferences, allow the user to select among additional scripts that can be used without alerting. The default can be based on the user's locale.
    3. In preferences, allow the user to choose a backward compatibility setting; see *Section 2.9.1, Backward Compatibility*.

D. Alerts
    1. If the user agent maintains a domain whitelist for the user, and the domain name is in the whitelist, allow it and skip the remaining items in this section. (The domain whitelist can take into account the documented policies of the registry as per *Section 2.10.4, Recommendations for Registries* .)
    2. If the visual appearance of a link does not match the end location, alert the user.
    3. If the domain name does not satisfy the requirements of the user

preferences (such as the Restriction Level), alert the user.

4. If the domain name contains any letters confusable with syntax characters, alert the user.
5. If there is a whitelist, and the domain name is visually confusable with a whitelist domain name, but not identical to it (after nameprep), alert the user.
6. If any label in the domain name is a whole-script or a mixed-script confusable, alert the user.

2.11.4 Recommendations for Registries

The following recommendations are for registries in dealing with identifiers such as domain names. The term "Registry" is to be interpreted broadly, as any agency that sets the policy for which identifiers are accepted.

Thus the .com operator can impose restrictions on the 2nd level domain label, but if someone registers *foo.com*, then it is up to them to decide what will be allowed at the 3rd level (for example, *bar.foo.com*). So for that purpose, the owner of *foo.com* is treated as the "Registry" for the 3rd level (the *bar*). Similarly, the owner of a domain name is acting as an internal registry in terms of the policies for the non-domain name portions of a URL, such as *banking* in *http://bar.foo.com/banking.* Thus the following recommendations still apply.

For information on the confusable tests mentioned below, see *Section 4, Confusable Detection* in *UTS #39: Unicode Security Mechanisms* [UTS39].

A. Publicly document the Restriction Level being enforced. For IDN, the Restriction Level is not to be higher than Level 4: that is, no characters can be outside of the *General Security Profiles for Identifiers* in *Section 3, Identifier Characters* in *UTS #39: Unicode Security Mechanisms* [UTS39].
B. Publicly document the enforcement policy on confusables: whether two domain names are allowed to be single-script or mixed script confusables.
C. If there are any pre-existing exceptions to A or B, then document them also.
D. Define an IDN registration in terms of both its Nameprep-Normalized Unicode representation (the *output format*) and its Punycode representation.

2.11.5 Registrar Recommendations

The following recommendations are for registrars in dealing with domain names. The term "Registrar" is to be interpreted broadly, as any agency that presents a UI for registering domain names, and allows users to see whether a name is registered. The same entity may be both a Registrar and Registry.

A. When a user's name is (or would be) rejected by the registry for security reasons, show the user the reason for rejection (such as the existence of an already-registered confusable).

3 Non-Visual Security Issues

There are a number of exploits based on misuse of character encodings. Some of these are fairly well-known, such as buffer overflows in conversion, while others are not. Many are involved in the common practice of having a 'gatekeeper' for a system. That gatekeeper checks incoming data to ensure that it is safe, and passes only safe data through. Once in the system, the other components assume that the data is safe. A problem arises when a component treats two pieces of text as identical—typically by canonicalizing them to the same form—but the gatekeeper only detected that one of them was unsafe.

For example, suppose that strings containing the letters "delete" are sensitive internally, and that therefore a gatekeeper checks for them. If some process casefolds "DELETE" *after* the gatekeeper has checked, then the sensitive string can sneak through. While many programmers are aware of this, they may not be aware that the same thing can happen with other transformations, such as an NFKC transformation of "Ⓓⓔⓛⓔⓣⓔ" into "delete".

These gatekeeper problems can also happen with charset converters. Where a character in a source string cannot be expressed in a target string, it is quite common for charset converters to have a "fallback conversion", picking the next best conversion. For example, when converting from Unicode to Latin-1, the character "ⓔ" cannot be expressed exactly, and the converter may fall back to "e". This can be used for the same kind of exploit. Unfortunately, some charset converter APIs, such as in Java, do not allow such fallbacks to be turned off. This is not only a problem for security, but also for other kinds of processing. For example, when converting an XML or HTML page, a character such as "ⓔ" missing from the target charset must be represented by an NCR such as &#x24D4; instead of using a lossy converter. Where possible, using Unicode instead of other charsets avoids many of these kinds of problems.

3.1 UTF-8 Exploits

There are three equivalent encoding forms for Unicode: UTF-8, UTF-16, and UTF-32. UTF-8 is commonly used in XML and HTML; UTF-16 is the most common in program APIs; and UTF-32 is the best for representing single characters. While these forms are all equivalent in terms of the ability to express Unicode, the original usage of UTF-8 was open to a canonicalization exploit.

Originally, Unicode forbade the *generation* of "non-shortest form" UTF-8, but not the *interpretation* of "non-shortest form" UTF-8. This was fixed in Unicode 3.0, because security issues can arise when software does interpret the non-shortest forms. For example:

- Process *A* performs security checks, but does not check for non-shortest forms.
- Process *B* accepts the byte sequence from process *A*, and transforms it into UTF-16 while interpreting non-shortest forms.
- The UTF-16 text may then contain characters that should have been filtered out by process *A*.

For example, the backslash character "\" can often be a dangerous character to let through a gatekeeper, because it can be used to access different directories. Thus a gatekeeper might specifically prevent it from getting through. The backslash is represented in UTF-8 as the byte sequence <5C>. However, as a non-shortest form, backslash could also be represented as the byte sequence<C1 9C>. When a gatekeeper does not check for non-shortest form, this situation can lead to a severe security breach.

To address this issue, the Unicode Technical Committee modified the definition of UTF-8 in Unicode 3.1 to forbid conformant implementations from interpreting non-shortest forms for BMP characters, and clarified some of the conformance clauses.

3.1.1 Ill-Formed Subsequences

Suppose that a UTF-8 converter is iterating through input UTF-8 bytes, converting to an output character encoding. If the converter encounters an ill-formed UTF-8 sequence it can treat it as an error in a number of different ways, including substituting a character like U+FFFD, SUB, "?", or SPACE. However, it *must not* consume any valid successor bytes. For example, suppose we have the following sequence:

X = <... 41 **C2** 3E 42 ... >

This sequence overall is ill-formed, because it contains an ill-formed substring, namely the <**C2**>. That is, there is no substring of X containing the **C2** byte which matches the specification for UTF-8 in Table 3-7 of Unicode 5.2 [Unicode]. The UTF-8 converter can stop at the **C2** byte, or substitute a character or sequence like U+FFFD and continue. However, it must not consume the **3E** byte if it continues. That is, it is acceptable to convert X to "...**A >B**...", but not acceptable to convert X to **"...A B..."** (that is, deleting the >).

Consuming a subsequent byte (such as **3E** above) is not only non-conformant; it can lead to security breaches. For example, suppose that a web page is constructed with user input. The user input is filtered to catch problem attributes such as onMouseOver. However, incorrect conversion can defeat that filtering by removing important syntax characters like > in HTML attribute values. Take the following string, where "✗" indicates a bare **C2** byte:

```
<span style=width:100%✗> onMouseOver=doBadStuff()...
```

When this is converted with a bad UTF-8 converter, the **C2** would cause the > character to be consumed, and the HTML served up would be of the following form, allowing for a cross-site scripting attack:

```
<span style=width:100% onMouseOver=doBadStuff()...
```

For more information on how to handle ill-formed subsequences, see "Constraints on Conversion Processes" in *Section 3.9, Unicode Encoding Forms* in Unicode 5.2 [Unicode].

3.1.2 Substituting for Ill-Formed Subsequences

If characters *are* to be substituted for ill-formed subsequences, it is important that those characters be relatively safe.

- Deletion (substituting the empty string) can be quite nasty, because it joins characters that would have been separate (such as on MouseOver).
- Substituting characters that are valid syntax for constructs such as file names has similar problems. For example, the '.' can be very problematic.
  - U+FFFD is usually unproblematic, because it is designed expressly for this kind of purpose. That is, because it does not have syntactic meaning in programming languages or structured data, it will typically just cause a failure in parsing. Where the output character set is not Unicode, though, this character may not be available.
  - Where U+FFFD is not available, a common alternative is "?". While this character may occur syntactically, it appears to be less subject to attack than most others.

UTF-16 converters that do not handle isolated surrogates correctly are subject to the same type of attack, although historically UTF-16 converters have generally handled these well.

3.2 Text Comparison (Sorting, Searching, Matching)

The UTF-8 exploit is a special case of a general problem. Security problems may arise where a user and a system (or two systems) compare text differently. For example, this happens where text does not compare as users expect. See the discussions in *UTS#10: Unicode Collation Algorithm* [UTS10], especially Section 1.

A system is particularly vulnerable when two different implementations of the same protocol use different mechanisms for text comparison, such as the comparison as to whether two identifiers are equivalent or not.

Assume a system consists of two modules: a user registry and the access control. Suppose that the user registry does not use NamePrep, while the access control module does. Two situations can arise:

1. The user with valid access rights to a certain resource actually cannot access it, because the binary representation of user ID used for the user registry differs from the one specified in the access control list. This situation is not a major security concern—because the person in this situation cannot access the protected resource.

2. The opposite case creates a security hole: a new user whose ID is NamePrep-equivalent to another user's in the directory system can get the access right to a protected resource.

For example, a fundamental standard, [LDAP], used to be subject to this problem; thus steps were taken to remedy this in later versions.

There are some other areas to watch for. Where these are overlooked, it may leave a system open to the text comparison security problems.

1. Normalization is context dependent; do not assume NFC(x + y) = NFC(x) + NFC(y).

2. There are ***two*** binary Unicode orders: code point/UTF-8/UTF-32 and UTF-16 order. In the latter, U+10000 **<** U+E000 (because U+10000 = D800 DC00).
3. Avoid using non-Unicode charsets where possible. IANA / MIME charset names are ill-defined: vendors often convert the same charset different ways. For example, in Shift-JIS the value 0x5C converts to ***either*** U+005C ***or*** U+00A5 depending on the vendor, resulting in different, unrelated characters with unrelated glyphs. See:
    - http://www.w3.org/TR/japanese-xml/
    - http://icu.sourceforge.net/charts/charset/
4. When converting charsets, *never* simply omit characters that cannot be converted; at least substitute U+FFFD (when converting to Unicode) or 0x1A (when converting to bytes) to reduce security problems. See also [UTS22].
5. Regular expression engines use character properties in matching. They may vary in how they match, depending on the interpretation of those properties. Where regex matching is important to security, ensure that the regular expression engine conforms to the requirements of [UTS18], and uses an up-to-date version of the Unicode Standard for its properties.

Transitivity is crucial to correct functioning of sorting algorithms. Transitivity means that if a < b and b < c then a < c. It means that there cannot be any cycles: a < b < c < a.

A lack of transitivity in string comparisons may cause security problems, including denial-of-service attacks. As an example of a failure of transitivity, consider the following pseudocode:

```
int compare(a,b) {
  if (isNumber(a) && isNumber(b)) {
    return numberComparison(a,b);
  } else {
    return textComparison(a,b);
  }
}
```

The code seems straightforward, but produces the following non-transitive result:

"12" < "12a" < "2" < "12"

For the first two comparisons, one of the values is not a number, therefore both values are compared as text. For the last two, both are numbers, and compared numerically. This breaks transitivity because a cycle is introduced.

The following pseudocode illustrates one way to repair the code, by sorting all numbers before all non-numbers:

```
int compare(a,b) {
  if (isNumber(a)) {
    if (isNumber(b)) {
      return numberComparison(a,b);
    } else {
      return -1; // a is less than b, since a is a number and b isn't
    }
  } else if (isNumber(b)) {
    return 1;    // b is less than a, since b is a number and a isn't
  } else {
    return textComparison(a,b);
  }
}
```

Therefore, for complex comparisons, such as language-sensitive comparison, it is important to test for transitivity thoroughly.

3.3 Buffer Overflows

Some programmers may rely on limitations that are true of ASCII or Latin-1, but fail with general Unicode text. These can cause failures such as buffer overruns if the length of text grows. In particular:

1. Strings may expand in casing: Fluß → FLUSS → fluss. The expansion factor may change depending on the UTF as well.
2. Programmers assume that NFC always composes, and thus is the same or shorter length than the original source. However, some characters *decompose* in NFC. The expansion factor may change depending on the UTF as well.
3. *Table 9, Maximum Expansion Factors* illustrates the expansions for case operations and normalization. These factors are for a particular version of Unicode: they should be recomputed for the particular version of Unicode being used.
   - The very large factors in the case of NFKC and NFKD are due to some extremely rare characters. Thus algorithms can use much smaller expansion factors for the typical cases as long as they have a fallback process that accounts for the possibility of these characters in data.
   - As of Unicode 5.0, a *Stream-Safe Text Format* was added to *UAX #15: Unicode Normalization Forms [UAX15]* . This format allows protocols to limit the number of characters that they need to buffer in handling normalization.
4. When performing character conversion, text may grow or shrink, sometimes substantially. Always account for that possibility in processing.

Table 9.
Maximum Expansion Factors

Operation	UTF	Factor	Sample	
Lower	8	1.5X	Ⱥ	U+023A
	16, 32	1X	A	U+0041
Upper/Title/Fold	8, 16, 32	3X	ΐ	U+0390
Operation	**UTF**	**Factor**	**Sample**	
NFC	8	3X	𝅘𝅥𝅮	U+1D160
	16, 32	3X	שּׁ	U+FB2C
NFD	8	3X	ΐ	U+0390

	16, 32	4X	ᾂ	U+1F82
NFKC/NFKD	8	11X	ﷺ	U+FDFA
	16, 32	18X		

3.4 Property and Character Stability

The Unicode Consortium Stability Policies [Stability] limit the ways in which the standards developed by the Unicode Consortium can change. These policies are intended to ensure that text encoded in one version of the Unicode Standard remains valid and unchanged in later versions. In many cases, the constraints imposed by these stability policies allow implementers to simplify support for particular features of Unicode, with the assurance that their implementations will not be invalidated by a later update to Unicode.

Implementations should not make assumptions beyond what is documented in the Stability Policies. For example, some implementations assumed that no new decomposable characters would be added to Unicode. The actual restriction is slightly looser: that decomposable characters will not be added if their decompositions were already in Unicode. It is therefore possible to add a decomposable character *if* one of the characters in its decomposition is also new in that version of Unicode. For example, decomposable Balinese characters were added to the standard in Version 5.0, which caused some implementations to break.

Similarly, some applications assumed that all Chinese characters were three bytes in UTF-8. Thus once a string was known to be all Chinese, iteration through the string could take the form of simply advancing an offset or pointer by three bytes. This assumption proved incorrect and caused implementations to break when Chinese characters were added on Plane 2, requiring 4-byte representations in UTF-8.

Making such unwarranted assumptions can lead to security problems. For example, advancing uniformly by three bytes for Chinese will corrupt the interpretation of text, leading to problems like those mentioned in *Section 3.1.1, Ill-Formed Subsequences*. Implementers should thus be careful to only depend on the documented stability policies.

An implementation may need to make certain assumptions for performance —assumptions that are not guaranteed by the policies. In such a case, it is recommended to at least have unit tests that detect whether those assumptions have become invalid when the implementation is upgraded to a new version of Unicode. That allows the problem to be detected and code to be revised if the assumption is invalidated.

3.5 Deletion of Code Points

In some versions prior to Unicode 5.2, conformance clause C7 allowed the deletion of noncharacter code points:

> C7. When a process purports not to modify the interpretation of a valid coded character sequence, it shall make no change to that coded character sequence other than the possible replacement of character sequences by their canonical-

equivalent sequences ***or the deletion of noncharacter code points*.**

Whenever a character is invisibly deleted (instead of replaced), such as in this older version of C7, it may cause a security problem. The issue is the following: A gateway might be checking for a sensitive sequence of characters, say "delete". If what is passed in is "deXlete", where X is a noncharacter, the gateway lets it through: the sequence "deXlete" may be in and of itself harmless. However, suppose that later on, past the gateway, an internal process invisibly deletes the X. In that case, the sensitive sequence of characters is formed, and can lead to a security breach.

The following is an example of how this can be used for malicious purposes.

> <a href=“java**\uFEFF**script:alert("XSS")>

3.6 Secure Encoding Conversion

In addition to handling Unicode text safely, character encoding conversion also needs to be designed and implemented carefully in order to avoid security issues.

3.6.1 Illegal Input Byte Sequences

When converting from a multi-byte encoding, a byte value may not be a valid trailing byte, in a context where it follows a particular leading byte. For example, when converting UTF-8 input, the byte sequence E3 80 22 is malformed because 0x22 is not a valid second trailing byte following the leading byte 0xE3. Some conversion code may report the three-byte sequence E3 80 22 as one illegal sequence and continue converting the rest, while other conversion code may report only the two-byte sequence E3 80 as an illegal sequence and continue converting with the 0x22 byte which is a syntax character in HTML and XML (U+0022 double quote). Implementations that report the 0x22 byte as part of the illegal sequence can be exploited for cross-site-scripting (XSS) attacks.

Therefore, an illegal byte sequence must not include bytes that encode valid characters or are leading bytes for valid characters.

The following are safe error handling strategies for conversion code dealing with illegal multi-byte sequences. (An illegal single/leading byte does not pose this problem.)

1. Stop with an error. Do not continue converting the rest of the text.
2. In a reported illegal byte sequence, do not include any non-initial byte that encodes a valid character or is a leading byte for a valid sequence.
3. Report the first byte of the illegal sequence as an error and continue with the second byte.

Strategy 1 is the simplest, but in many cases it is desirable to convert as much of the text as possible. For example, a web browser will usually replace a small number of illegal byte sequences with U+FFFD each and display the page as best it can. Strategy 3 is the next simplest but can lead to multiple U+FFFD or other error handling artifacts for what is a single-byte error.

Strategy 2 is the most natural and fits well with an assumption that most errors are not due to physical transmission corruption but due to truncated multi-byte sequences from improper string handling. It also avoids going back to an earlier byte stream position in most cases.

Converters for single-byte encodings are unaffected by any of these issues. Nor are converters for the Character Encoding Schemes UTF-16 and UTF-32 and their variants affected, because they are not really byte-based encodings: they are often "converted" via memcpy(), at most with a byte swap, so a converter needs to always deliver pairs or quads of bytes.

3.6.2 Some Output For All Input

Character encoding conversion must also not simply skip an illegal input byte sequence. Instead, it must stop with an error or substitute a replacement character (such as U+FFFD (�) REPLACEMENT CHARACTER) or an escape sequence in the output. (See also *Section 3.5 Deletion of Code Points*.) It is important to do this not only for byte sequences that encode characters, but also for unrecognized or "empty" state-change sequences. For example:

- An illegal or unrecognized ISO-2022 designation or escape sequence.
- Pairs of SI/SO without text characters between them.
- ISO-2022 shift sequences without text characters before the next shift sequence. The formal syntaxes for HZ and most CJK ISO-2022 variants require at least one character in a text segment between shift sequences. Security software written to the formal specification may not detect malicious text (for example, "delete" with a shift-to-double-byte then an immediate shift-to-ASCII in the middle).

3.7 Enabling Lossless Conversion

There is a known problem with file systems that use a legacy charset. When a Unicode API is used to find the files in a directory, the return value is a list Unicode file names. Those names are used to access the files through some other API. There are two possible problems:

- One of the file names is invalid according to the legacy charset converter. For example, it is an SJIS string consisting of bytes <E0 30>.
- Two of the file names are mapped to the same Unicode string by the legacy charset converter.

These problems come up in other situations besides file systems as well. One common source of the problem is a byte string valid in one charset that is converted according to a different charset. For example, the byte string <E0 30> is invalid in SJIS, but is perfectly meaningful in Latin-1, representing "à0".

One possible solution is to enable all charset converters to losslessly (reversibly) convert to Unicode. That is, any sequence of bytes can be converted by each charset converter to a Unicode string, and that Unicode string would be converted back to exactly that original sequence of bytes by the converter. This precludes, for example, the charset converter's mapping two different unmappable byte sequences to `U+FFFD`

(�) REPLACEMENT CHARACTER, because the original bytes could not be recovered. It also precludes having "fallbacks" (see http://unicode.org/reports/tr22/): cases where two different byte sequences map to the same Unicode sequence.

3.7.1 PEP 383 Approach

PEP 383 takes this approach. It enables lossless conversion to Unicode by converting all "unmappable" sequences to a sequence of one or more isolated surrogate code points. That is, each unmappable byte's value is a code point whose value is 0xDC00 plus byte value. With this mechanism, every maximal subsequence of bytes that can be reversibly mapped to Unicode by the charset converter is so mapped; any intervening subsequences are converted to a sequence of high surrogates. The result is a Unicode String, but not a well-formed UTF sequence.

For example, suppose that the byte 81 is illegal in charset *n*. When converted to Unicode, PEP 383 represents this as U+D881. When mapped back to bytes for charset *n*, it turns back into the byte 81. This allows the source byte sequence to be reversibly represented in a Unicode String, no matter what the contents. If this mechanism is applied to a charset converter that has no fallbacks from bytes to Unicode, then the charset converter becomes reversible (from bytes to Unicode to bytes).

This only works when the Unicode String is converted back with the very same charset converter that was used to convert from bytes. For more information on PEP 383, see http://python.org/dev/peps/pep-0383/.

3.7.2 Notation

The following notation is used in the rest of this section:

- B2Un is the bytes-to-Unicode converter for charset n
- U2Bn is the Unicode-to-bytes converter for charset n
- An *invalid* byte is one that would be mapped by a PEP to a high surrogate, because it is part of a sequence that is not reversibly mappable. The context of the byte is important: for example, the byte 81 alone might be unmappable, while an 81 followed by a 40 is valid.

3.7.3 Security

Unicode implementations have been subject to a number of security exploits centered around ill-formed encoding, such as http://blogs.technet.com/srd/archive/2009/05/18/more-information-about-the-iis-authentication-bypass.aspx. Systems making incorrect use of a PEP 383-style mechanism are subject to such an attack.

Suppose that the source byte stream is <A B X D>, and that according to the charset converter being used (n), X is an invalid byte. B2Un transforms the byte stream into Unicode as <G Y H>, where Y is an isolated surrogate. U2Bn maps back to the correct original <A B X D>. This is the intended usage of PEP 383.

The problem comes when that Unicode sequence is converted back to bytes by a different charset converter *m*. Suppose that U2Bm maps Y into a valid byte representing

"/", or any one of a number of other security-sensitive characters. That means that converting <G Y H> via U2Bm to bytes, and back to Unicode results in the string "G/Y", where the "/" did not exist in the original.

This violates one of the cardinal security rules for transformations of Unicode strings: creating a character where no valid character previously existed. This was at the heart of the "non-shortest form" security exploits. A gatekeeper watches for suspicious characters. It does not see Y as one of them, but past the gatekeeper, a conversion of U2Bm followed by B2Um results in a suspicious character where none previously existed.

There is a suggested solution for this. A converter would map an isolated surrogate Y onto a byte stream only when the resulting byte would be an *illegal* byte. If not, then an exception would be thrown, or a replacement byte or byte sequence must be used instead (such as the SUB character). For details, see *Section 3.7.5* Safely Converting to Bytes . This replacement would be similar to what is used when trying to convert a Unicode character that cannot be represented in the target encoding. This strategy preserves the ability to round-trip when the same encoding is used, but prevents security attacks. *Note that simply deleting Y in the output is not an option, because that is also open to security exploits.*

When used as intended in Python, PEP 383 appears unlikely to present security problems. According to information from the author:

- PEP 383 is only intended for use with ASCII-based charsets.
- Only bytes >= 128 will be transformed to D8xx or back.
- The combination of these factors means that no ASCII-repertoire characters (which represent the most serious problems for security) would ever be generated.
- The primary use of PEP 383 is in file systems, where the Unicode String resulting from PEP 383 is only converted back to bytes on the same system, using the same charset converter.

However, if PEP 383 is used more generally by applications, or similar systems are used more generally, security exploits are possible.

3.7.4 Interoperability

Using isolated surrogates (D8xx) as the way to represent the unconvertible bytes appears harmless at first glance. However, it presents certain interoperability and security issues. Such isolated surrogates are not well-formed. Although they can be represented in a Unicode String, they are not supported by conformant UTF-8, UTF-16, or UTF-32 converters or implementations. This may cause interoperability problems, because many systems replace incoming ill-formed Unicode sequences by replacement characters. It may also cause security problems. Although strongly discouraged for security reasons, some implementations may delete the isolated surrogates, which can cause a security problem when two separated substrings become adjacent.

There are different alternatives:

1. Use 256 private-use code points, somewhere in the ranges F0000..FFFFD or

100000..10FFFD. This would probably cause the fewest security and interoperability problems. There is, however, some possibility of collision with other uses of private-use characters.

2. Use pairs of noncharacter code points in the range FDD0..FDEF. These are "super" private-use characters, and are discouraged for general interchange. The transformation would take each nibble of a byte Y, and add to FDD0 and FDE0, respectively. However, noncharacter code points may be replaced by U+FFFD (�) REPLACEMENT CHARACTER by some implementations, especially when they use them internally. *(Again, incoming characters must never be deleted, because that can cause security problems.)*

3.7.5 Safely Converting to Bytes

The following describes how to safely convert a Unicode buffer U1 to a byte buffer B1 when the D8xx convention is used.

- Convert from Unicode buffer U1 to byte buffer B1.
- If there were any D8XX's in U1
  - Convert back to Unicode buffer U2 (according to the same Charset C1)
  - If U1 != U2, throw an exception.

This approach is simple, and sufficient for the vast majority of implementations because the frequency of D8xx's will be extremely low. Where necessary, there are a number of different optimizations that can be used to increase performance.

3.8 Idempotence

idempotence is a property of a function, whereby repeated application of that function produces the same result. That is: f(f(x)) = f(x). Some functions have this property, such as f(x) := |x|, while others do not, such as f(x) := x+1.

Properties that are expected to be idempotent—but actually aren't—can represent severe problems for security. For more information, see the Unicode Security FAQ.

Appendix A Script Icons

Table 10, Sample Script Icons shows sample icons that can be used to represent scripts in user interfaces. They are derived from from the *Last Resort Font*, which is available on the Unicode site [LastResort]. While the Last Resort Font is organized by Unicode block instead of by script, the glyphs from that font can also be used to represent scripts. This is done by picking one of the possible glyphs whenever a script spans multiple blocks.

Table 10. Sample Script Icons

Arabic	Armenian	Bengali
Bopomofo	Braille	Buginese

Buhid	Canadian Aboriginal	Cherokee
Coptic	Cypriot	Cyrillic
Deseret	Devanagari	Ethiopic
Georgian	Glagolitic	Gothic
Greek	Gujarati	Gurmukhi
Hangul	Han	Hanunoo
Hebrew	Hiragana	Latin
Lao	Limbu	Linear B
Kannada	Katakana	Kharoshthi
Khmer	Mongolian	Myanmar
Malayalam	Ogham	Old Italic
Old Persian	Oriya	Osmanya
New Tai Lue	Runic	Shavian
Sinhala	Syloti Nagri	Syriac
Tagalog	Tagbanwa	Tai Le
Tamil	Telugu	Thaana
Thai	Tibetan	Tifinagh
Ugaritic	Yi	

Special cases

Common	Inherited	

Appendix B Language-Based Security

It is very hard to determine exactly which characters are used by a language. For example, English is commonly thought of as having letters A-Z, but in customary practice many other letters appear as well. For examples, consider proper names such as "Zoë", words from the Oxford English Dictionary such as "coöperate", and many foreign words in common use: "René", ‘naïve’, ‘déjà vu’, ‘résumé’, and so on.Thus the problem with restricting identifiers by language is the difficulty in defining exactly what that implies. See the following definitions:

> **Language**: Communication of thoughts and feelings through a system of arbitrary signals, such as voice sounds, gestures, or written symbols. Such a system including its rules for combining its components, such as words. Such a system as used by a nation, people, or other distinct community; often contrasted with

> dialect. *(From American Heritage, Web search)*
>
> **Language**: The systematic, conventional use of sounds, signs, or written symbols in a human society for communication and self-expression. Within this broad definition, it is possible to distinguish several uses, operating at different levels of abstraction. In particular, linguists distinguish between language viewed as an act of speaking, writing, or signing, in a given situation […], the linguistic system underlying an individual's use of speech, writing, or sign […], and the abstract system underlying the spoken, written, or signed behaviour of a whole community. *(David Crystal, An Encyclopedia of Language and Languages)*
>
> **Language** is a finite system of arbitrary symbols combined according to rules of grammar for the purpose of communication. Individual languages use sounds, gestures, and other symbols to represent objects, concepts, emotions, ideas, and thoughts…
>
> Making a principled distinction between one language and another is usually impossible. For example, the boundaries between named language groups are in effect arbitrary due to blending between populations (the dialect continuum). For instance, there are dialects of German very similar to Dutch which are not mutually intelligible with other dialects of (what Germans call) German.
>
> Some like to make parallels with biology, where it is not always possible to make a well-defined distinction between one species and the next. In either case, the ultimate difficulty may stem from the interactions between languages and populations. *http://en.wikipedia.org/wiki/Language, September 2005*

The Unicode Common Locale Data Repository (CLDR) supplies a set of exemplar characters per language, the characters used to write that language. Originally, there was a single set per language. However, it became clear that a single set per language was far too restrictive, and the structure was revised to provide auxiliary characters, other characters that are in more or less common use in newspapers, product and company names, and so on. For example, auxiliary set provided for English is: [áà éè íì óò úù âêîôû æœ äëïöüÿ āēīōū ăĕĭŏŭ åø çñß]. As this set makes clear, the frequency of occurrence of a given character may depend greatly on the domain of discourse, and it is difficult to draw a precise line; instead there is a trailing off of frequency of occurrence.

In contrast, the definitions of writing systems and scripts are much simpler:

> **Writing system**: A determined collection of characters or signs together with an associated conventional spelling of texts, and the principle therefore. *(extrapolated from Daniels/Bright: The World's Writing Systems)*
>
> **Script**: A collection of symbols used to represent textual information in one or more writing systems.

Writing systems and scripts only relate to the written form of the language and do not require judgment calls concerning language boundaries. Therefore security considerations that relate to written form of languages are often better served by using the concept of writing system and/or script.

> **Note:** A writing system uses one or more scripts, plus additional symbols such as punctuation. For example, the Japanese writing system uses the scripts Hiragana, Katakana, Kanji (Han ideographs), and sometimes Latin.

Nevertheless, language identifiers are extremely useful in other contexts. They allow cultural tailoring for all sorts of processing such as sorting, line breaking, and text formatting.

> **Note:** As mentioned below, language identifiers (called language tags), may contain information about the writing system and can help to determine an appropriate script.

As explained in the *Section 6.1, Writing Systems* of [Unicode], scripts can be classified in various groups: Alphabets, Abjads, Abugidas, Logosyllabaries, Simple or Featural Syllabaries. Those classifications, in addition to historic evidence, makes it reasonably easy to arrange encoded characters into script classes.

The set of characters sharing the same script value determines a script set. The script value can be easily determined by using the information available in *UAX #24: Unicode Script Property*. No such concept exists for languages. It is generally not possible to attach a single language property value to a given character. Similarly, it is not possible to determine the exact repertoire of characters used for the written expression of most common languages.

Creating "safe character sets" is an important goal in a security context, and it would appear that the characters used in a language is an obvious choice. However, because of the indeterminate set of characters used for a language, it is typically more effective to move to the higher level, the script, which can be more easily specified and tested.

Customarily, languages are written in a small number of scripts. This is reflected in the structure of language tags, as defined by BCP47 "Tags for the Identification of Languages", which are the industry standard for the identification of languages. Languages that require more than one script are given separate language tags. See http://www.iana.org/assignments/language-subtag-registry.

The CLDR also provides a mapping from languages to scripts which is being extended over time to more languages. *Table 11, CLDR Script Mappings* provides examples of the association between language tags and default scripts. (CLDR also provides other information about scripts, such as the most likely language for each script, and the most likely script for each language, plus script metadata.)

Table 11. CLDR Script Mappings

Language tag	Script(s)	Comment
en	Latin	Content in ‘en’ is presumed to be in Latin script, unless where explicitly marked
az-	Cyrillic	Azeri in Cyrillic script used in Azerbaijan
az-Latn-AZ	Latin	Azeri in Latin script used in Azerbaijan

az	Latin,	Azeri as used generically, can be Latin or Cyrillic
ja	Han,	Japanese as used in Japan or elsewhere

The strategy of using scripts works extremely well for most of the encoded scripts because users are either familiar with the entirety of the script content, or the outlying characters are not very confusable. There are however a few important exceptions, such as the Latin and Han scripts. In those cases, it is recommended to exclude certain technical and historic characters except where there is a clear requirement for them in a language.

Lastly, text confusability is an inherent attribute of many writing systems. However, if the character collection is restricted to the set familiar to a culture, it is expected by the user, and he or she can therefore weigh the accuracy of the written or displayed text. The key is to (normally) restrict identifiers to a single script, thus vastly reducing the problems with confusability. For example, in Devanagari, the letter *aa*: आ can be confused with the sequence consisting of the letter a अ followed by the vowel sign aa ◌ा. However, this is a confusability a Hindi speaking user may be familiar with, as it relates to the structure of the Devanagari script.

In contrast, text confusability that crosses script boundary is completely unexpected by users within a culture, and unless some mitigation is in place, it will create significant security risk. For example, the Cyrillic small letter п ("pe") is undistinguishable from the Greek letter π in at least some fonts, and the confusion is likely to be unknown to users in cultural context using either script. Restricting the identifier to either wholy Greek or wholy Cyrillic will usually avoid this issue.

Acknowledgments

Mark Davis and Michel Suignard authored the bulk of the text, under the direction of the Unicode Technical Committee. Steven Loomis and other people on the ICU team were very helpful in developing the original proposal for this technical report. Thanks also to the following people for their feedback or contributions to this document or earlier versions of it: Julie Allen, Stéphane Bortzmeyer, Roger Costello, Douglas Davidson, Martin Dürst, Peter Edberg, Asmus Freytag, Deborah Goldsmith, Paul Hoffman, Patrick L. Jones, Peter Karlsson, Gervase Markham, Eric Muller, Erik van der Poel, Michael van Riper, Marcos Sanz, Alexander Savenkov, Markus Scherer, Dominikus Scherkl, Dave Thompson, Kenneth Whistler, and Yoshito Umaoka.

References

[CharMod] Character Model for the World Wide Web 1.0: Fundamentals
http://www.w3.org/TR/charmod/

[DCore] Derived Core Properties
http://www.unicode.org/Public/UNIDATA/DerivedCoreProperties.txt

[DemoConf] http://unicode.org/cldr/utility/confusables.jsp

[DemoIDN] http://unicode.org/cldr/utility/idna.jsp

[DemoIDNChars] http://unicode.org/cldr/utility/list-unicodeset.jsp?a=\p{age%3D3.2}-\p{cn}-\p{cs}-\p{co}&abb=on&g=uts46+idna+idna2008

[Display] Display Problems?
http://www.unicode.org/help/display_problems.html

[FAQSec] Unicode FAQ on Security Issues
http://www.unicode.org/faq/security.html

[ICANN] ICANN Documents:

Internationalized Domain Names
http://www.icann.org/en/topics/idn/

The IDN Variant Issues Project
http://www.icann.org/en/topics/new-gtlds/idn-vip-integrated-issues-23dec11-en.pdf

[IDNA2003] The IDNA2003 specification is defined by a cluster of IETF RFCs:

- IDNA [RFC3490]
- Nameprep [RFC3491]
- Punycode [RFC3492]
- Stringprep [RFC3454].

[IDNA2008] The IDNA2008 specification is defined by a cluster of IETF RFCs:

- Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework
  http://tools.ietf.org/html/rfc5890
- Internationalized Domain Names in Applications (IDNA) Protocol
  http://tools.ietf.org/html/rfc5891
- The Unicode Code Points and Internationalized Domain Names for Applications (IDNA)
  http://tools.ietf.org/html/rfc5892

- Right-to-Left Scripts for Internationalized Domain Names for Applications (IDNA)
  http://tools.ietf.org/html/rfc5893

There are also informative documents:

- Internationalized Domain Names for Applications (IDNA): Background, Explanation, and Rationale
  http://tools.ietf.org/html/rfc5894
- The Unicode Code Points and Internationalized Domain Names for Applications (IDNA) - Unicode 6.0
  http://tools.ietf.org/html/rfc6452

[IDN-Demo] http://unicode.org/cldr/utility/idna.jsp

[IDN-FAQ] http://www.unicode.org/faq/idn.html

[IDN-Demo] ICU (International Components for Unicode) IDN Demo
http://demo.icu-project.org/icu-bin/icudemos

[Feedback] Reporting Form
http://www.unicode.org/reporting.html
For reporting errors and requesting information online.

[LastResort] Last Resort Font
http://unicode.org/policies/lastresortfont_eula.html
(See also http://www.unicode.org/charts/lastresort.html)

[LDAP] Lightweight Directory Access Protocol (LDAP): Internationalized String Preparation
http://www.rfc-editor.org/rfc/rfc4518.txt

[NFKC_Casefold] The Unicode property specified in [UAX44], and defined by the data in DerivedNormalizationProps.txt (search for "NFKC_Casefold").

[Reports] Unicode Technical Reports
http://www.unicode.org/reports/
For information on the status and development process for technical reports, and for a list of technical reports.

[RFC1034] P. Mockapetris. "DOMAIN NAMES - CONCEPTS AND FACILITIES", RFC 1034, November 1987.

http://ietf.org/rfc/rfc1034.txt

[RFC1035] P. Mockapetris. "DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION", RFC 1034, November 1987.
http://ietf.org/rfc/rfc1035.txt

[RFC1535] E. Gavron. "A Security Problem and Proposed Correction With Widely Deployed DNS Software", RFC 1535, October 1993
http://ietf.org/rfc/rfc1535.txt

[RFC3454] P. Hoffman, M. Blanchet. "Preparation of Internationalized Strings ("stringprep")", RFC 3454, December 2002.
http://ietf.org/rfc/rfc3454.txt

[RFC3490] Faltstrom, P., Hoffman, P. and A. Costello, "Internationalizing Domain Names in Applications (IDNA)", RFC 3490, March 2003.
http://ietf.org/rfc/rfc3490.txt

[RFC3491] Hoffman, P. and M. Blanchet, "Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)", RFC 3491, March 2003.
http://ietf.org/rfc/rfc3491.txt

[RFC3492] Costello, A., "Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)", RFC 3492, March 2003.
http://ietf.org/rfc/rfc3492.txt

[RFC3743] Konishi, K., Huang, K., Qian, H. and Y. Ko, "Joint Engineering Team (JET) Guidelines for Internationalized Domain Names (IDN) Registration and Administration for Chinese, Japanese, and Korean", RFC 3743, April 2004.
http://ietf.org/rfc/rfc3743.txt

[RFC3986] T. Berners-Lee, R. Fielding, L. Masinter. "Uniform Resource Identifier (URI): Generic Syntax", RFC 3986, January 2005.
http://ietf.org/rfc/rfc3986.txt

[RFC3987] M. Duerst, M. Suignard. "Internationalized Resource Identifiers (IRIs)", RFC 3987, January 2005.
http://ietf.org/rfc/rfc3987.txt

[Stability] Unicode Character Encoding Stability Policy
http://www.unicode.org/standard/stability_policy.html

[UCD] Unicode Character Database.
http://www.unicode.org/ucd/
For an overview of the Unicode Character Database and a list of its associated files.

[UCDFormat] UCD File Format
http://www.unicode.org/reports/tr44/#Format_Conventions

[UAX9] UAX #9: The Bidirectional Algorithm
http://www.unicode.org/reports/tr9/

[UAX15] UAX #15: Unicode Normalization Forms
http://www.unicode.org/reports/tr15/

[UAX24] UAX #24: Unicode Script Property
http://www.unicode.org/reports/tr24/

[UAX31] UAX #31, Identifier and Pattern Syntax
http://www.unicode.org/reports/tr31/

[UAX44] UAX #44:Unicode Character Database
http://www.unicode.org/reports/tr44/

[Unicode] The Unicode Standard
For the latest version, see:
http://www.unicode.org/versions/latest/

[UTS10] UTS #10: Unicode Collation Algorithm
http://www.unicode.org/reports/tr10/

[UTS18] UTS #18: Unicode Regular Expressions
http://www.unicode.org/reports/tr18/

[UTS22] UTS #22: Character Mapping Markup Language (CharMapML)
http://www.unicode.org/reports/tr22/

[UTS39] UTS #39: Unicode Security Mechanisms
http://www.unicode.org/reports/tr39/

[UTS46] Unicode IDNA Compatibility Processing
http://www.unicode.org/reports/tr46/

[Versions] Versions of the Unicode Standard
http://www.unicode.org/standard/versions/

For information on version numbering, and citing and referencing the Unicode Standard, the Unicode Character Database, and Unicode Technical Reports.

Modifications

The following summarizes modifications from the previous revisions of this document.

Revision 14

- *Section 1.1* *Structure*
  - Added a note on the broad use of the term “URL”, and replaced some instances elsewhere of URI and IRI.
- *Section 2.8.1* *Punycode Spoofs*
  - Added a description of how the display of Punycode URLs instead of Unicode can be worse for spoofing.
- *Section 2.10* *Restriction Levels and Alerts*
  - Add a second example of an alert, for mixed scripts.
- *Section 2.11.2* *Recommendations for Programmers*
  - Added note on the use of Catalan in identifiers.

Revision 13

- *Section 3.1.1* *Ill-Formed Subsequences*
  - Fixed various typos.
- *Section 3.2* *Text Comparison (Sorting, Searching, Matching)*
  - Added description of issues with transitivity
- *Section 3.7.1* *PEP 383 Approach*
  - Removed the incorrect term 'high' on 'surrogate'.
- *Section 3.8* *Idempotence*
  - Added pointer to article about idempotence.
- Fleshed out table of contents, fixed links, and incorrect numbering of sections in 2.9-2.10.
- Changed references to point to the http://www.unicode.org/faq/security.html for links that might change.

Revision 12 being a proposed update, only changes between revisions 11 and 13 are noted here.

Revision 11

- Moved definition of Restriction Levels to UTS #39
- Fixed reported typos, and updated references.

Revision 10 being a proposed update, only changes between revisions 9 and 11 are noted here.

Revision 9

- Added table numbers and explicit references to tables in the text.
- Expanded the introduction to Section 3 somewhat.
- Removed Appendices A, B, D, E, and F, and renumbered the other Appendices.
- Moved external references to the FAQ
- Cleaned up references to UTS39 and UTS46
- Removed former Appendix F.
- Added Section 3.6, Secure Encoding Conversion.
- Added Section 3.7, Enabling Lossless Conversion.
- Removed old Section 3.6, Recommendations
- Clarified *Section 3.5, Deletion of Code Points*
- Miscellaneous other editorial changes.

Revision 8 being a proposed update, only changes between revisions 7 and 9 are noted here.

Revision 7

- Added explanation of UTF-8 over-consumption attack in 3.1 UTF-8 Exploits
- Added subsection of 2.8.2 Mapping and Prohibition describing the Unicode 5.1 changes in identifiers.
- Added 3.4 Property and Character Stability
- Updated Unicode reference.
- Broke 3.1.1 into two sections, adding header 3.1.2: Substituting for Ill-Formed Subsequences, with some small wording changes around it. In particular, pointed to *Appendix E. Conformance Changes to the Standard* in Unicode 5.1.
- Added 3.5 Deletion of Noncharacters
- Added before Sample Country Registries: "These are only for illustration: the exact sets may change over time, so the particular authorities should be consulted rather than relying on these contents. Some registrars now also offer machine-readable formats."
- Minor editing

Revision 6 being a proposed update, only changes between revisions 4 and 7 are noted here.

Revision 4

- Moved the contents of *Appendix A Identifier Characters*, *Appendix B, Confusable Detection*, and *Appendix D Mixed Script Detection* to the new [UTS39]. The appendices remain (to avoid renumbering), but simply point to the new locations. Changed references to point to the new sections in [UTS39].
- Alphabetized *Appendix C. Script Icons.*
- Added *Appendix G. Language-Based Security.*
- Changed the "highlighting" of the core domain name to the whole domain name in

Section 2.6, Syntax Spoofing.

- Replaced *Section 2.9.4 Recommendations for Registries* based on the UTC decisions.
- Removed the contents of *Appendix E. Future Topics*, incorporating material to address the issues in *Section 3.2, Text Comparison, Section 3.3, Buffer Overflows* , and a few other places in the document.
- Minor editing

Revision 3

- Cleaned up references
- Added Related Material section
- Add section on Casefolded Format
- Refined recommendations on single-script confusables
- Reorganized introduction, and reversed the order of the main sections.
- Retitled the main sections
- Restructured the recommendations for Visual Security
- Added more examples
- Incorporated changes for user feedback
- Major restructuring, especially appendices. Moved data files and other references into the references, added section on confusables, scripts, future topics, revised the identifiers section to point at the newer data file.
- Incorporated changes for all the editorial notes: shifted some sections.
- Added sections on bidi, appendix F.
- Revised data files

Revision 2

- Moved recommendations to separate section.
- Added new descriptions, recommendations.
- Pointed to draft data files.

Revision 1

- Initial version, following proposal to UTC.
- Incorporated comments, restructured, added To Do items.