L2/10-022

Technical Reports

Proposed Update Unicode Technical Report #36

UNICODE SECURITY CONSIDERATIONS

Version	4 (draft 3)
Authors	Mark Davis (markdavis@google.com), Michel Suignard (michel@suignard.com)
Date	2010-02-04
This Version	http://www.unicode.org/reports/tr36/tr36-8.html
Previous Version	http://www.unicode.org/reports/tr36/tr36-7.html
Latest Version	http://www.unicode.org/reports/tr36/
~~Latest Working Draft~~	~~http://www.unicode.org/draft/reports/tr36/tr36.html~~
Revision	8

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*This is a **draft** document which may be updated, replaced, or superseded by other documents at any time.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].

To allow access to the most recent work of the Unicode security subcommittee on this document, the "Latest Working Draft" link in the header points to the latest working-draft document under development.

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.

But because Unicode contains such a large number of characters, and because it 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, since clever Javascript or ActiveX can work around that. (And users may 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://citibank.com/.... They see that it is, and use their password. But what they saw was wrong — it is actually going to a spoof site with a fake "citibank.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. But 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. ...[Paypal].

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.

> ~~**Note**: Some of the examples below use Unicode characters which some browsers will not show, or may not show in a way that illustrates the problem. For more information about improving the display in your browser, see [Display].~~

For examples and background information, see the References, including the Related Material. For information on possible future topics, see *Appendix E. Future Topics*. See also the Unicode FAQ on *Security Issues* [FAQSec].

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 + descent is from 9 to 12 pixels for most scripts, somewhat larger for scripts, such as Japanese, where the users typically select 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. Note that 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 at all 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 [UTS39].

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

2.1 Internationalized Domain Names

Visual spoofing is an especially important subject given the recent introduction of Internationalized Domain Names (IDN). 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. So IDN represents 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. For example, 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". Any place where strings are used as identifiers is subject to this kind of spoofing.

IDN provides a good starting point for a discussion of visual spoofing, and will be used as the focus for the remaining part of this section. However, the concepts and recommendations discussed here can be generalized to the use of other types of identifiers. For background information on identifiers, see UAX #31: *Identifier and Pattern Syntax* [UAX31].

Certain parts of domain names are still required to be in ASCII, and thus not subject to the visual spoofing issues discussed here. For example, the top-level domain names (.com, .ru, etc.) are currently always ASCII (this may change in the future, however).

Fortunately the design of IDN prevents a huge number of spoofing attacks. All conformant users of IDN 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 of the possibilities for visual spoofing by mapping away a large number of visually confusable characters and sequences. For example, characters like the half-width 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". But the compatibility normalization will convert any sequence of "a" plus "¨" into the regular "ä".

Thus you can not spoof an *a-umlaut* with *a + umlaut*; it simply results in the same domain name. See the example *Safe Domain Names* below. The String column shows the actual characters; the UTF-16 shows the underlying encoding, while 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).

[Review Note: The tables will be numbered, and centered, with added anchors.]

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 + umlaut
1b	ät.com	**00E4** 0074 002E 0063 006F 006D	xn--t-zfa.com	But it ends up being identical to the composed form, in IDNA

Note: The ICU demo at [IDN-Demo] can be used to demonstrate the results of processing different domain names. That demo was also used to get the ACE values shown here.

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.

The IDN processing also removes case distinctions by performing a *case folding* to reduce characters to a lowercase form. This is also useful for avoiding spoofing problems, since characters are generally more distinctive in their lowercase forms. That means that people can focus on just the lowercase characters.

This focus on lowercase letters only really helps for *Internationalized* Domain Names, because of two factors: First, the IDNA operation ToASCII() will map to lowercase if and only if the label contains some non-ASCII character. Thus ToASCII("paypaI.com") (where the 'I' is a capital 'i') produces no change.

Secondly, domain names are case-insensitive, but [RFC1034] and [RFC1035]~~, as clarified by [DNS-Case]~~, introduce the concept of case preservation. Thus if someone queries the DNS for "paypaI.com", and the DNS contains information for "paypai.com", that information is delivered, but the answer from the DNS will be the original "paypaI.com".

There are some cases where people will want to see differences preserved in display. For more information, and information about characters allowed in IDN, see [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 so as to cover user expectations (just as one may register both "analyze.com" and "analyse.com" to cover variant spellings).

Although normalization and case-folding prevent many possible spoofing attacks, visual spoofing can still occur with many Internationalized Domain Names. 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 *user agent* side (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* [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 equally responsive to security issues
- due to the decentralized nature of DNS, registries do not control subdomains being established beyond the domain name registered

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 existing data to be mapped to Unicode 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 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 leads to a significant number of spoofing possibilities using characters from different scripts. For example, a domain name can be spoofed by using a Greek omicron instead of an 'o', as in example 1a in the following table.

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 *Appendix D. Mixed Script Detection*.

Cyrillic, Latin, and Greek represent special challenges, since the number of common glyphs shared between them is so high, as can be seen from *Section 4. Confusable Detection* [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 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 following examples, all in the same script. In each numbered case, the strings will look identical or close to identical in most browsers

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	año.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.

As mentioned above, 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. Example:

Combining Mark Order Spoofing

	String	UTF-16	ACE	Comments
1a	□□□.com	101C **102D** 102F	xn--gjd8ag.com	Reorders two combining marks
1b	□□□.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 certain sequences of characters. These are characters or sequences of characters that should be visually distinguishable, but do not appear that way. 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, in roughly 30% of the fonts the glyphs 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. But as this example 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.

Inadequate Rendering Support

	String	UTF-16	ACE	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	

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.

There are a number of characters in Unicode that 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. 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 deform 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, since it is powerful enough to change the appearance of, say "$100.00" on the screen to "$200.00" when printed.

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

As with many other cases, this is 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 (or bidi in short). The relationship between the memory representation of the text (logical order) and the display appearance (visual order) of bidi text is governed by the 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, both the IDN and IRI specifications require 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.

In addition, the IRI specification extends those requirements to other components of an IRI, not just the host name labels. Not respecting them would result in insurmountable visual confusion. A large part of the confusability in reading an IRI containing bidi characters is created by the weak or neutral directionality property of many IRI/URI delimiters such as '/', '.', '?' which makes them change directionality depending on their surrounding characters. For example, in example #1 in the table below, the dots following each label are colored the same as that label. Notice that the placement of that following punctuation may vary.

Bidi Examples

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

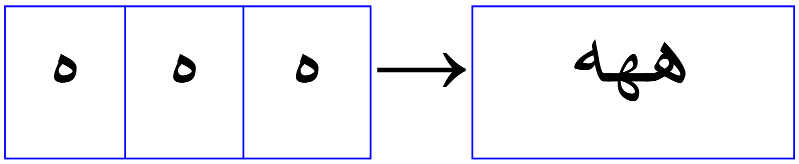
- Never allow BIDI override characters.
- As much as possible, avoid mixing right-to-left and left-to-right characters in a single host name

- When right-to-left characters are used, limit the usage of left-to-right characters to well-known cases such as TLD names and URI/IRI scheme names (such as http, ftp, mailto, etc...)
- 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.
- Reverse-bidi (visual order -> storage order) can be used to detect 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.

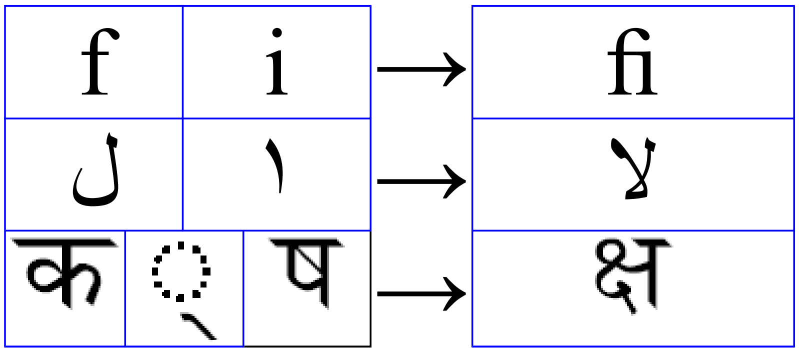
2.5.1 Complex Scripts

In complex scripts such as Arabic and South Asian scripts, characters may change shape according to the surrounding characters:

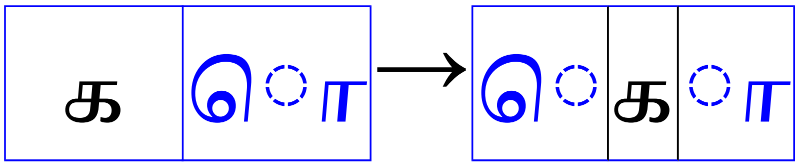
1. Glyphs may change shape depending on their surroundings:



2. Multiple characters may produce a single glyph:



3. A single character may produce multiple glyphs:



In such cases, two characters may be visually distinct in a stand-alone form, but might not be distinct in a particular context.

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 even identified, and the visual interpretation of the characters depends entirely on the font, and is completely unconnected with the underlying characters. Luckily such font-encodings are seldom used, and decreasing in use with the growth of Unicode.

2.6 Syntax Spoofing

Spoofing syntax characters can be even worse than regular characters. 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 of the following table may appear to be in the domain **macchiato.com**, but is actually in a particular subzone of the domain **bad.com**.

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

Other syntax characters, if there are visual confusables, can be similarly spoofed, as in lines 2 through 4. Many — but not all — of these cases, such as U+2024 (․) ONE DOT LEADER are disallowed by Nameprep [RFC3491].

Of course, a spoof fooling the user into thinking that the domain name is the first part of the URL does not require internationalized domain names. 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 "most significant part first". The following hypothetical display would be harder to spoof: the fact that it is "com.bad" is not as easily lost.

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

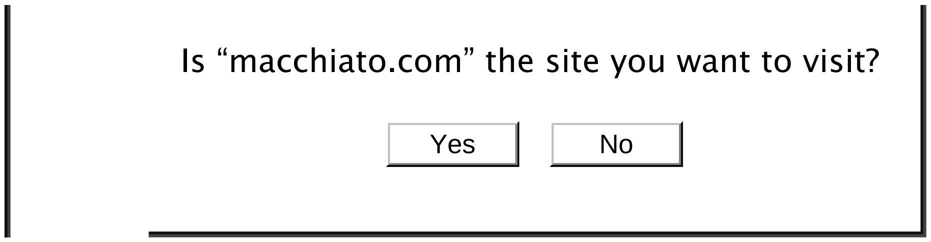
But that would be an impossible change at this point: it is long past the time when such a radical change could have been made. However, a possible solution is to always visually distinguish the domain, for example:

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 log into 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, as described below.

2.6.1 Missing Glyphs

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

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 IDN, because private use characters are excluded by NamePrep.

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 {0}. 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}. While the sets taken as a whole are different in shape, individual digits may have the same shapes as digits from other scripts, even digits of different values. For example, the string ৪୨ is visually confusable with 89 (at small sizes), but actually has the numeric value 42.

Where software interprets the numeric value of a string of digits without detecting that the digits are from different scripts, it is possible to generate such spoofs.

2.7a 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 browser is using a browser on the pre-2010 international domain name specification or the post-2010 specification. For more information on this topic, see [UTS46] and [IDN_FAQ].

[Review Note: The sections will be renumbered consecutively.]

2.8 Techniques

This section lists techniques that can be used in reducing the risks of visual spoofing. These techniques are referenced by *Section 2.10 Recommendations*.

2.8.1 Case-Folded Format

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

However, there is one particular situation where the pure case-folded format of a string as defined by the standard is not desired. 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 case folding algorithm needs to map both of them together (so that transitivity is maintained), only one of them appears in the case-folded form.

When the case-folded format of a Greek string is to be displayed to the user, it should be processed so as to choose the proper form for the small sigma, depending on the context. The test for the context is provided in Table 3-15 of [Unicode]. It is the test for Final_Sigma, where C represents the character σ. Basically, 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.

Final Sigma Handling (from Table 3-15)

Context	Description	Regular Expressions	
Final_Sigma	C is preceded by a sequence consisting of a cased letter and a case-ignorable sequence, and C is not followed by a sequence consisting of a case ignorable sequence and then a cased letter.	Before C:	\p{cased} (\p{case-ignorable})*
		After C:	! ((\p{case-ignorable})* \p{cased})

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].

[Review Note: The section numbers, links and text will be adjusted as necessary to follow the ed committee style guidelines.]

2.8.2 Mapping and Prohibition

There are two techniques to reduce the risk of spoofing that can usefully be applied to identifiers: mapping and prohibition. IDNA uses both of these. A number of characters are included in Unicode for compatibility. What is called *Compatibility Normalization* (NFKC) can be used to map these characters to the regular variants (this is what is done in IDNA). For example, a half-width Japanese *katakana* character ｶ is mapped to the regular character カ. Additional mappings can be added beyond compatibility mappings, for example, IDNA 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 ₨ to "rs"

In addition, characters may be prohibited. For example, IDNA prohibits *space* and *no-break space* (U+00A0). Instead, for example, of removing a ZERO WIDTH JOINER, or mapping ₨ 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, then 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, 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, in version 5.1 of the Unicode Standard the recommendations for identifiers have been 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.

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 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*.

2.9 Restriction Levels and Alerts

The Restriction Levels 1-5 are defined below for use in implementations. These place restrictions on the use of identifiers according to the appropriate Identifier Profile as specified in *Section 3. Identifier Characters* [UTS39], and the determination of script as specified in *Section 4. Confusable Detection* [UTS39]*.* For IDNA, the particular Identifier Profile will be one of the two specified in *Section 3.1. General Security Profile for Identifiers* [UTS39].

1. **ASCII-Only**
    - All characters in each identifier must be ASCII
2. **Highly Restrictive**
    - All characters in each identifier must be from a single script, or from the combinations:
      ASCII + Han + Hiragana + Katakana;
      ASCII + Han + Bopomofo; or
      ASCII + Han + Hangul
    - No characters in the identifier can be outside of the Identifier Profile
    - Note that this level will satisfy the vast majority of Latin-script users.

3. **Moderately Restrictive**
    - Allow *Latin* with other scripts except *Cyrillic*, *Greek*, *Cherokee*
    - Otherwise, the same as **Highly Restrictive**
4. **Minimally Restrictive**
    - Allow arbitrary mixtures of scripts, e.g. Ωmega, Teχ, HλLF-LIFE, Toys-Я-Us.
    - Otherwise, the same as **Moderately Restrictive**
5. **Unrestricted**
    - Any valid identifiers, including characters outside of the Identifier Profile, e.g. I♥NY.org

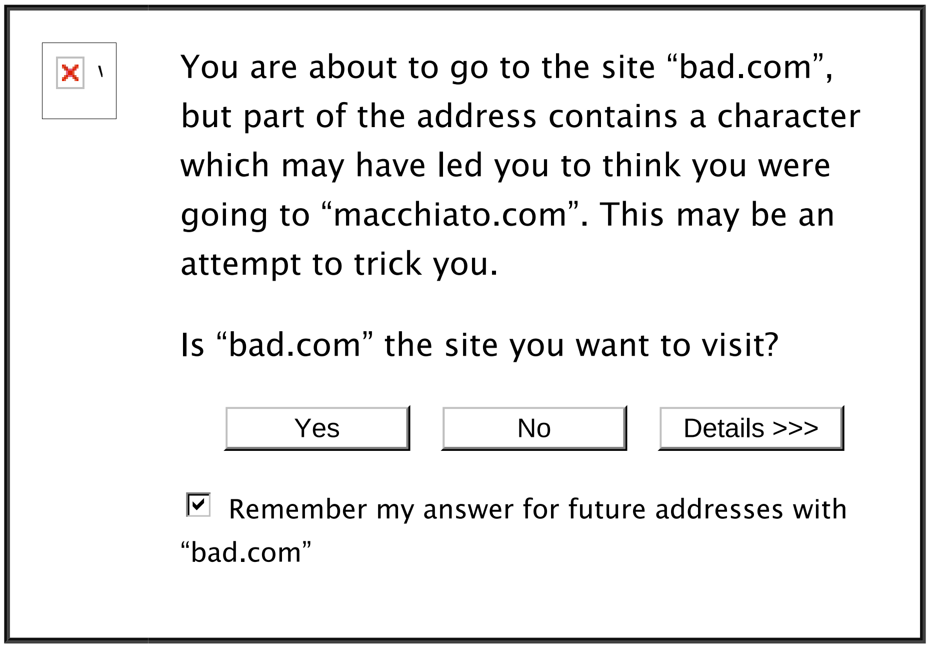
An appropriate alert should be generated if an identifier fails to satisfy the Restriction Level chosen by the user. Depending on the circumstances and the level difference, the form of such alerts could be minimal, such as special coloring or icons (perhaps with a tool-tip for more information); or 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 C. 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 (but not all) languages that customarily use the Latin script, there is probably little need to mix in other scripts. That is not necessary the case for other languages. Because of the widespread commercial use of English and other Latin-based languages (such as "خدمة RSS"), it is quite common to have instances of Latin (especially ASCII) in text that principally consists of other scripts.

Section 3. Identifier Characters [UTS39] provides for two profiles of identifiers that could be used in Restriction Levels 1 through 4. The strict profile is the recommended one. If the lenient one is also allowed, the user should have a choice in preferences, so that there is some way to limit the levels to using the strict input 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:



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”

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

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 upper 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 know to the user agent to be 'safe'.

2.9.1 Backwards 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 are continually being added to the Unicode Standard that may be valid for identifiers. The confusable information may add more characters as visually confusable over time.

There may also be cases where characters are no longer recommended for inclusion in identifiers, and more information becomes available about them. Thus the identifier profile may become more restrictive in a future version, for some characters. Of course, once identifiers are registered they cannot be withdrawn, but new proposed identifiers that contain such characters can be denied. A user-agent should give users a preference setting that essentially uses the union of the old and new identifier profiles in determining the Restriction Levels.

2.10 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 G. Language-Based Security.

Instead, the recommendations call for combination of string preprocessing 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].

2.10.1 User Recommendations

A. Use browsers, mail clients and software in general 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. ~~For more information, see *Appendix F. Country-Specific IDN Restrictions*.~~
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 not automatically provided by the registry. For how to detect confusables, see *Section 4. Confusable Detection* [UTS39].
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*. If you are using documents with embedded fonts (aka web fonts), be aware that the content on printed form (the one, for example, that you may sign) can be different than what you see on the screen.

2.10.2 General Programmer Recommendations

A. When parsing numbers: detect digits of mixed (or whole but 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* [UTS39].

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 itself 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 your own 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 C. Script Icons*.

2.10.3 User Agent Recommendations

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* [UTS39]*.* If the user can see the case-folded form, use the lowercase-only confusable mappings; otherwise use the broader mappings.

A. Follow Section 2.10.2 General Programmer Recommendations.
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 Case-Folded 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 backwards compatibility setting; see *Section 2.9.1* Backwards 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* Registry Recommendations.)
2. If the visual appearance of a link (if it looks like a URL) 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.

2.10.4 Registry Recommendations

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 hold. (In particular, StringPrep and the IDN Security Profiles should be used.)

For information on the confusable tests mentioned below, see *Section 4. Confusable Detection* in [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 *IDN Security Profiles for Identifiers* in [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 ACE representation.

2.10.5 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 why the name was rejected (such as the existence of an already-registered confusable).

3. Non-Visual Security Issues

A common practice is to have 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 — while the gatekeeper only detected that one of them was unsafe.

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.

Up to *The Unicode Standard, Version 3.0* the *generation* of "non-shortest form" UTF-8 was forbidden, as was the *interpretation* of illegal sequences, but not the interpretation of what was called the "non-shortest form". Where software does interpret the non-shortest forms, security issues can arise. 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, since 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. For more information, see [Related Material].

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 sequence

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

This sequence overall is ill-formed, because it contains an ill-formed substring, 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 5.1 [Unicode]. The UTF-8 converter can stop at the **C2** byte, or substitute a character or sequence like U+FFFD and continue. But it must not consume the **3E** byte if it does continue. That is, it is ok to convert X to ...**A** >B..., but not ok to convert X to **...A B...** (that is, deleting the >).

Consuming any subsequent byte 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. But 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 precisely how to handle ill-formed subsequences, see *E. Conformance Changes to the Standard* in Unicode 5.1 [Unicode].

3.1.2 Substituting for Ill-Formed Subsequences

Note that 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, since it joins characters that would have been separate (eg on MouseOver).
- Substituting characters that are valid syntax for constructs such as file names has similar problems. The '.' for example can be very problematic.
    - U+FFFD is usually unproblematic, because it is designed expressly for this kind of purpose. That is, because it doesn't 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 don't handle isolated surrogates correctly are subject to the same type of attack, although historically UTF-16 converters have had 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, where text does not compare as users expect, this can cause security problems. See the discussions in UTS#10: Unicode Collation Algorithm [UTS10], especially Sections 1 1.5.

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 is different from the one specified in the access control list. This situation is actually not too bad from a security standpoint – because the person in this situation cannot access the protected resource.

2. In the opposite case, it's 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, is subject to this problem; thus steps ~~were~~ taken to remedy this [LDAP]. ~~In the meantime, since you cannot rely on the implementation of any particular LDAP server, so you should wrap the user registration module so as to StringPrep the user IDs for registration, and then use exactly the same normalization logic to maintain the access control list.~~

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; don't assume NFC(x + y) = NFC(x) + NFC(y).

2. There are ***two*** binary Unicode orders: code point/UTF-8/UTF-32 and UTF16 order. In the latter, U+10000 < U+E000 (since 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.
   ► 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 you are using conforms to the requirements of [UTS18], and uses an up-to-date version of the Unicode Standard for its properties.

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. Table 3.3 contains the current maximum expansion factors for each casing operations, for each UTF.
2. People 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. Table 3.3 *Maximum Expansion Factors in Unicode 5.0* contains the maximal expansion factors for each normalization form in each UTF. These are calculated for Unicode 5.0; this may change in the future.
    - The very large factors in the case of NFKC/D 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.
    - In Unicode 5.0, a new *Stream-Safe Text Format* is has been 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.

3. When doing character conversion, text may grow or shrink, sometimes substantially. Always account for that possibility in processing.

Table 3.3
Maximum Expansion Factors in Unicode 5.0

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	صلى الله عليه وسلم	U+FDFA
	16, 32	18X		

3.4 Property and Character Stability

The Unicode Consortium Stability Policies [Stability] limits 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 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 the standard, with the assurance that their implementations will not be invalidated by a later update to the standard.

Implementations should not make assumptions beyond what is documented on these pages. For example, some implementations assumed that no new decomposable characters would be added to Unicode. The actual restriction is slightly looser: roughly that decomposable characters won't be added if their decompositions were already in Unicode. So a decomposable character can be added if one of the characters in its decomposition is also new. For example, decomposable Balinese characters were added to the standard in Version 5.0.

Similarly, some applications assumed that all Chinese characters were 3 bytes in UTF-8. Thus once a string was known to be all Chinese, then iteration through the string could take the form of simply advancing an offset or pointer by 3 bytes. This assumption proved incorrect and caused problems for implementations 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 3 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 — ones 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 code to be revised if that were to happen.

3.5 Deletion of Code Points

In some versions prior to Unicode 5.2, conformance clause C7 allowed the deletion of non-character 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***.

~~Although the last phrase permits the deletion of noncharacter code points, for security reasons, they only should be removed with caution.~~

Whenever a character is invisibly deleted (instead of replaced), 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. But 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\uFEFFscript: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.

As illustrated, it is not safe to include in an illegal byte sequence 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. Do not include in a reported illegal byte sequence 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.

Conversion of single-byte encodings is unaffected by any of these strategies.

These section also does not apply to converters for the Character Encoding Schemes UTF-16 and UTF-32 and their variants 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

Similar to the considerations in 3.5 Deletion of Noncharacters, character encoding conversion must also not simply skip an illegal input byte sequence but rather stop with an error or substitute a Replacement Character or an escape sequence etc. in the output. It is important to do this not only for byte sequences that encode characters, but also for unrecognized or "empty" state-change sequences.

Examples:

- 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 (e.g. "delete" with a shift-to-double-byte then an immediate shift-to-ASCII in the middle).

[Review Note: This section may be combined with 3.1.1.]

3.7 Enabling Lossless Conversion

[Review note: The following is a rough draft and needs thorough review. TODO: Change references to use the standard style; number sections properly and enter into TOC.]

There is a known problem with file systems that use a legacy charset. When you use a Unicode API to find the files in a directory, you typically get back a list of Unicode file names. You use those names 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. A common source is when a byte string that is valid in one charset is converted by a different charset's converter. For example, the byte string <E0 30> that is invalid in SJIS is perfectly meaningful in Latin-1, representing "à0".

A possible solution to this 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 will be converted back to exactly that original sequence of bytes by that converter. This precludes, for example, the charset converter's mapping two different unmappable byte sequences to U+FFFD (�) REPLACEMENT CHARACTER, since 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

The basic idea of PEP 383 is to be able to enable lossless conversion to Unicode by converting all "unmappable" sequences to a sequence of one or more isolated high surrogate code points: that is, each code point's value is 0xD800 plus the corresponding unmappable 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 is 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*), then that 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).

Note that 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 (or is part of a sequence that is) not reversibly mappable. Note that the context of the byte is important: 81 alone might be unmappable, while 81 followed by an 40 is valid (see http://demo.icu-project.org/icu-bin/convexp?conv=ibm-943_P15A-2003)

3.7.3 Security

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

Suppose 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>. That 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, for example, at the heart of the "non-shortest form" security exploits. A gatekeeper is watching for suspicious characters. It doesn't 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.

The suggested solution for this is that a converter can only map an isolated surrogate Y onto a byte stream 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 Safely Converting to Bytes, below. This replacement would be similar to what is used when trying to convert a Unicode character that cannot be represented in the target encoding. That preserves the ability to round-trip when the same encoding is used, but prevents security attacks. *Note that simply not represented (deleting) Y in the output is not an option, since 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, and where the Unicode resulting from PEP 383 is only ever 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

The choice of isolated surrogates (D8xx) as the way to represent the unconvertible bytes appears clever at first glance. However, it presents certain interoperability and security issue. Such isolated surrogates are not well formed. Although they can be represented in Unicode Strings, they are not supported by conformant UTF-8, UTF-16, or UTF-32 converters or implementations. This may cause interoperability problems, since 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 substrings become adjacent that were previously separated.

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 non-character 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, since that can cause security problems.)*
3. One could also ask the Unicode Consortium to encode characters expressly for that purpose. That would be essentially 256 different versions of U+FFFD (�) REPLACEMENT CHARACTER. The downside

of this approach is that even if it were accepted, it would take a couple of years to get into the standard.

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.

[Review Note: Add a note about the following issues, although we are not including them in this version of the document.

- the problem with fallback substitutions (aka "Best Fit Mappings").
- the general problem with transforms (normalize, case-fold, etc) after validation.
- stronger advice that using UTF-8 in interchange avoids many of the problems with conversion.

]

~~3.6 Recommendations~~

[Review note: the following was a summary, but readers found it just confusing.]

A. ~~Ensure that all implementations of UTF-8 used in a system are conformant to the latest version of Unicode. In particular,~~
   A. ~~Always use the so-called "shortest form" of UTF-8~~

   B. ~~With UTF-8 (or UTF-16) conversion, never consume bytes from well-formed sequences as part of error handling~~
   C. ~~Avoid problematic substitutions for ill-formed substrings.~~
   D. ~~Never go outside of 0..10FFFF~~$_{\text{16}}$
   E. ~~Never use 5 or 6 byte UTF-8.~~

B. ~~Those designing a protocol should ensure that the text comparison operation is precisely defined, including the Unicode casing folding operation, and the normalization (NFKC) operation. Identifiers should be limited to those specified in *Section 3.1. General Security Profile for Identifiers* [UTS39].~~

C. ~~If a registration system does not precisely specify the comparison operation, a work-around is to wrap the user registration module so as to NamePrep the user IDs for registration, and then use exactly the same normalization logic to maintain the access control list.~~

D. ~~Be aware of the possible pitfalls with text comparison, removal of noncharacters, and buffer overflows; follow the recommendations in Sections 3.3 – 3.5.~~

[Review note: Sections A, B, D, and E have been empty since the last version. They will be removed and the remaining sections renumbered]

Appendix A. Identifier Characters

The mechanisms described in this section have been moved to [UTS39], Section 3.

Appendix B. Confusable Detection

The mechanisms described in this section have been moved to [UTS39], Section 4.

Appendix C. Script Icons

The following are 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.

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 D. Mixed Script Detection

The mechanisms described in this section have been moved to [UTS39], Section 5.

Appendix E. Future Topics

The former contents of this section have been incorporated into the document proper, or moved elsewhere.

~~Appendix F. Country-Specific IDN Restrictions~~

~~[Review note: because the links and information in this section need more frequent update, the plan is to move the information to http://www.unicode.org/faq/security.html.]~~

~~ICANN (Internet Corporation For Assigned Names and Numbers), among other tasks, is responsible for coordinating the management of the technical elements of the DNS to ensure universal resolvability. As such, after the IDNA RFCs were published in March 2003, ICANN and a cross-section of IDN-implementing registries published in June 2003 the first version of the "Guidelines for the Implementation of Internationalized Domain Names" [ICANN]. These guidelines include the following items:~~

- ~~strict compliance with the IDN RFCs~~
- ~~inclusion-based approach (characters not explicitly allowed are prohibited)~~
- ~~based on the need of a language or a group of languages~~
- ~~symbol characters, icons, dingbats, punctuations should not be included~~
- ~~consistent approach for language-specific registration policies~~
- ~~each domain label should be restricted to a single language or appropriategroup of languages~~

~~These guidelines have been endorsed by the .cn, .info, .jp, .org, and .tw registries. Furthermore, IANA (Internet Assigned Numbers Authority), following the ICANN guidelines about IDN, has created a registry for IDN Language Tables [IDNReg] which contains entries for:~~

- ~~.biz (German)~~
- ~~.info (German)~~
- ~~.jp (Japanese)~~
- ~~.kr (Korean)~~
- ~~.museum (Danish, Icelandic, Norwegian, Swedish, for more see [Museum])~~
- ~~.pl (Arabic, Hebrew, Greek, Polish)~~
- ~~.th (Thai)~~

~~Other registries have published their own IDN recommendations using various formats, such as the following.~~ ~~*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.*~~

~~Sample Country Registries~~

~~Brazil~~	~~.br~~	~~àáâã ç éê í óôõ úü~~	~~http://registro.br/faq/faq6.html#8~~
~~Denmark~~	~~.dk~~	~~åä æ é öø ü~~	~~http://www.difo.dk/regler/Tegn-01-01-2004.pdf~~
~~Chinese~~	~~.cn~~	~~(large list)~~	~~http://www.ietf.org/internet-drafts/draft-xdlee-ic expired in August 22nd 2005, so the link has becc successor.)~~
~~Germany~~	~~.de~~	~~àáâãäåāăą æ çćĉċč ďđ ð èéêëēĕėęě ĝğġģ ĥħ ìíîïĩīĭįı ĵ ķĸ ĺļľł ńņňŋñ òóôõöøōŏő~~	~~http://www.denic.de/en/domains/idns/liste.html~~

		~~œ ŕŗř śŝşš ţťŧ ŵ ùúûüũūŭůűų ýÿŷ źżž þ~~	
~~Hungary~~	~~.hu~~	~~á é í óöő úüű~~	~~http://www.domain.hu/domain/ekes/~~
~~Iceland~~	~~.is~~	~~á æ ð é í óö ú ý þ~~	~~https://www.isnic.is/~~
~~Latvia~~	~~.lv~~	~~ā č ē ģ ī ķ ļ ņ ō ŗ š ū ž~~	~~http://www.nic.lv/DNS/~~
~~Lithuania~~	~~.lt~~	~~ą č ęė į š ųū ž~~	~~http://www.domreg.lt/lt/nutar/leistini_simboliai.p~~
~~Norway~~	~~.no~~	~~áàäå æ čç đ éèê ŋńñ óòôöø š ŧ ü ž~~	~~http://www.norid.no/domeneregistrering/idn/idn~~
~~Portugal~~	~~.pt~~	~~àáâã ç éê í óôõ ú~~	~~https://online.dns.pt/imagens/site/home_227/fot~~
~~Sweden~~	~~.se~~	~~åä ö~~	~~http://www.nic-se.se/teknik/programvara_idn.sht~~
~~Switzerland~~	~~.ch~~	~~àáâãäå æ ç èéêë ìíîï ð ñ òóôõöø œ ùúûü ýÿ þ~~	~~http://www.switch.ch/id/faq/idn.html (English, Fr~~

~~**Note**: When documents are published in their native language, the IDN additions to the basic ASCII DNS repertoire have been mentioned in parenthesis.~~

~~**Note**: Some of the country-based registries do not strictly follow the language-based approach recommended by ICANN because they cover a group of languages, such as in Switzerland or in Germany. Furthermore, two countries using the same language may differ in their list of additional characters (for example, Brazil and Portugal).~~

~~There are probably more country-specific IDN recommendations, so this enumeration is by no mean exhaustive. As of now, the output list from *Section 3. Identifier Characters* [UTS39] is a strict superset of all country-specific restricted IDN lists itemized above.~~

Appendix G. 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, proper or not, that are in common use: "René", 'naïve', 'déjà vu', 'résumé', etc… Thus the problem with restricting identifiers by language is the difficulty in defining exactly what that implies. The problem with using language identifier in a security approach derives from the complexity to define what a language is. 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*

For example, 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, etc. 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, (a) the frequency of occurrence of a given character may depend greatly on the domain of discourse, and (b) 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. *(Unicode 4.1.0 UAX #24)*

The simplification originates from the fact that 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 much 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. That classification, 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 the Unicode Standard Annex UAX#24 (Script Names). 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. Languages tend to be fluid; words are added or disappear, foreign words using new characters from the original script may be borrowed.

> **Note**: A well known example is English itself which is commonly considered to only use the Latin letters A to Z, while in fact the large borrowing from the French language has introduced words or expressions such as ‘naïve’, ‘déjà vu’, ‘résumé’, etc.

Note: There are a few cases where script and languages are tightly connected, like Armenian, Lao, etc…However, using scripts in these cases preserves the general model.

Creating ‘safe character sets’ is an important goal in a security context. The benefit is to create a collection of characters that are deemed familiar for a given cultural environment. Incorporating all characters necessary to express the written language associated with the culture is the obvious choice. However, because of the indeterminate set of characters used for a language, it is much more effective to move to the higher level, the script, which can be determinately 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 RFC 3066 "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. Examples can be found in http://www.iana.org/assignments/language-tags.

The proposed successor to RFC3066, which was approved by the IETF in November of 2005 (but has not yet been published), makes this relationship with scripts more explicit, and provides information as to which scripts are implicit for which languages. CLDR also provides a mapping from languages to scripts which is being extended over time to more languages. The following table below provides examples of the association between language tags and scripts.

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

az-Latn-AZ	Latin	Azeri in Latin script used in Azerbaijan
az	Latin, Cyrillic	Azeri as used generically, can be Latin or Cyrillic
ja or ja-JP	Han, Hiragana, Katakana	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 weight 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.

> *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 ा. But this is a confusability a Hindi speaking user may be familiar 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.

> Example: The Cyrillic small letter п ("pe") is undistinguishable from the Greek letter π (at least with some fonts), and the confusion is

likely to be unknown to users in cultural context using either script. Restricting the set to either Greek or Cyrillic will eliminate this issue.

Although a language identifier can uniquely determine a safe set of characters in some rare cases, it is preferable to use the script property as predicate on a given culture to determine the safe character set.

Acknowledgements

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: Stéphane Bortzmeyer, Douglas Davidson, Martin Dürst, Peter Edberg, Asmus Freytag, Deborah Goldsmith, Paul Hoffman, Peter Karlsson, Gervase Markham, Eric Muller, Erik van der Poel, Michael van Riper, Marcos Sanz, Alexander Savenkov, Markus Scherer, Dominikus Scherkl, Kenneth Whistler, and Yoshito Umaoka.

References

References not listed here may be found in http://www.unicode.org/reports/tr41/#UAX41.

[Bortzmeyer]	http://www.bortzmeyer.org/idn-et-phishing.html (machine tı at http://translate.google.com/translate?u=http%3A%2F%2Fwww.bortzmeyer.org%2Fidn-et-phishing.html)
[CharMod]	Character Model for the World Wide Web 1.0: Fundamentals http://www.w3.org/TR/charmod/
~~[Charts]~~	~~Unicode Charts (with Last Resort Glyphs)~~ ~~http://www.unicode.org/charts/lastresort.html~~ ~~See also:~~ ~~http://developer.apple.com/fonts/LastResortFont/~~ ~~http://developer.apple.com/fonts/LastResortFont/LastResort~~
[DCore]	Derived Core Properties http://www.unicode.org/Public/UNIDATA/DerivedCoreProper

[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% {cn}-\p{cs}-\p{co}&abb=on&g=uts46+idna+idna2008
[Display]	Display Problems? http://www.unicode.org/help/display_problems.html
[DNS-Case]	Donald E. Eastlake 3rd. "Domain Name System (DNS) Case Ins Clarification". Internet Draft, January 2005 http://www.ietf.org/internet-drafts/draft-ietf-dnsext-insensi
[FAQSec]	Unicode FAQ on Security Issues http://www.unicode.org/faq/security.html
[ICANN]	Guidelines for the Implementation of Internationalized Domai http://icann.org/general/idn-guidelines-20sep05.htm (These are in development, and undergoing changes)
~~[ICU]~~	~~International Components for Unicode~~ ~~http://site.icu-project.org/~~
[IDN-Demo]	http://unicode.org/cldr/utility/idna.jsp
[IDN-FAQ]	http://www.unicode.org/faq/idn.html
~~[idnhtml]~~	~~IDN Characters, categorized into different sets.~~ ~~idn-chars.html~~
~~[IDNReg]~~	~~Registry for IDN Language Tables~~ ~~http://www.iana.org/assignments/idn/~~ ~~Tables are found at:~~ ~~http://www.iana.org/assignments/idn/registered.htm~~
[IDN-Demo]	ICU (International Components for Unicode) IDN Demo http://demo.icu-project.org/icu-bin/icudemos
[Feedback]	Reporting Errors and Requesting Information Online http://www.unicode.org/reporting.html **Type of Message: Technical Report...**

[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): Internationalize Preparation http://www.rfc-editor.org/rfc/rfc4518.txt
~~[Museum]~~	~~Internationalized Domain Names (IDN) in .museum – Supporte~~ ~~Languages~~ ~~http://about.museum/idn/language.html~~
[NFKC_CaseFold]	The Unicode property specified in [UAX44], and defined by the DerivedNormalizationProps.txt (search for "NFKC_CaseFold").
[Paypal]	Beware the 'Paypal' scam http://news.zdnet.co.uk/internet/security/0,39020375,2080
[Reports]	Unicode Technical Reports http://www.unicode.org/reports/ *For information on the status and development process for te* *reports, and for a list of technical reports.*
[RFC1034]	P. Mockapetris. "DOMAIN NAMES - CONCEPTS AND FACILITIES 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 Deployed DNS Software", RFC 1535, October 1993 http://ietf.org/rfc/rfc1535.txt
[RFC3454]	P. Hoffman, M. Blanchet. "Preparation of Internationalized Stri ("stringprep")", RFC 3454, December 2002. http://ietf.org/rfc/rfc3454.txt
[RFC3490]	Faltstrom, P., Hoffman, P. and A. Costello, "Internationalizing Names in Applications (IDNA)", RFC 3490, March 2003. http://ietf.org/rfc/rfc3490.txt

[RFC3491]	Hoffman, P. and M. Blanchet, "Nameprep: A Stringprep Profile Internationalized Domain Names (IDN)", RFC 3491, March 200 http://ietf.org/rfc/rfc3491.txt
[RFC3492]	Costello, A., "Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)", RFC March 2003. http://ietf.org/rfc/rfc3492.txt
[RFC3743]	Konishi, K., Huang, K., Qian, H. and Y. Ko, "Joint Engineering T Guidelines for Internationalized Domain Names (IDN) Registra Administration for Chinese, Japanese, and Korean", RFC 3743, 2004. http://ietf.org/rfc/rfc3743.txt
[RFC3986]	T. Berners-Lee, R. Fielding, L. Masinter. "Uniform Resource Ide (URI): Generic Syntax", RFC 3986, January 2005. http://ietf.org/rfc/rfc3986.txt
[RFC3987]	M. Duerst, M. Suignard. "Internationalized Resource Identifiers 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*~~ ~~*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: Script Names~~ ~~http://www.unicode.org/reports/tr24/~~

~~[UAX31]~~	~~UAX #31, Identifier and Pattern Syntax~~ ~~http://www.unicode.org/reports/tr31/~~
~~[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/~~
[Unicode]	The Unicode Consortium. The Unicode Standard, Version 5.2. by: The Unicode Standard, Version 5.2 (Mountain View, CA: Th Consortium, 2009. ISBN 978-1-936213-00-9)
[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 referen Unicode Standard, the Unicode Character Database, and Unic Technical Reports.*

Related Material

The background information may also be useful.

[Review note: because the links and information in this section need more frequ update, this information will be moved to http://www.unicode.org/faq/security

Canonical Representation

- Microsoft Security Bulletin (MS00-078): Patch Available for 'Web Server Fol Traversal' Vulnerability
- INS: Microsoft IIS Unicode Exploit
- Creating Arbitrary Shellcode In Unicode Expanded Strings

Visual Spoofing

- The Homograph Attack
- PayPal alert! Beware the 'Paypal' scam
- gTLD Registry Constutuency: Potential of IDN for malicious abuse
- ICANN | Internationalized Domain Names
- ICANN Email Archives [idn-homograph]
- Multiple Browsers IDN Spoofing Test
- Decision Strategies and Susceptibility to Phishing
- Why Phishing Works
- Do Security Toolbars Actually Prevent Phishing Attacks
- Phishing Tips and Techniques

Modifications

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

Revision 8

- Draft 3
- Made modifications resulting from UTC discussion.
- Removed Appendix F [title].
- Draft 2
- Proposed Update of the document.
- Added section 3.x Secure Encoding Conversion.
- Added section 3.y Enabling Lossless Conversion to Unicode.
- Removed old section ~~3.6 Recommendations~~
- Clarified section 3.5 Deletion of Code Points
- Fixed References section.
- Misc other editorial changes.

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 *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 5 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 2.10.4 Registry Recommendations* 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 Case-Folded 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
