Technical Reports

L2/16-139

Proposed Update Unicode® Technical Standard #39

UNICODE SECURITY MECHANISMS

Version	9.0.0 (draft 3)
Editors	Mark Davis (markdavis@google.com), Michel Suignard (michel@suignard.com)
Date	2016-04-07
This Version	http://www.unicode.org/reports/tr39/tr39-12.html
Previous Version	http://www.unicode.org/reports/tr39/tr39-11.html
Latest Version	http://www.unicode.org/reports/tr39/
Latest Proposed Update	http://www.unicode.org/reports/tr39/proposed.html
Revision	12

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 document specifies mechanisms that can be used to detect possible security problems.

Status

list of current Unicode Technical Reports, see [Reports]. For more information about versions of the Unicode Standard, see [Versions].

Contents

---

1 Introduction

Unicode Technical Report #36, "Unicode Security Considerations" [UTR36] provides guidelines for detecting and avoiding security problems connected with the use of Unicode. This document specifies mechanisms that are used in that document, and can be used elsewhere. Readers should be familiar with [UTR36] before continuing. See also the Unicode FAQ on *Security Issues* [FAQSec].

2 Conformance

An implementation claiming conformance to this specification must do so in conformance to the following clauses:

C1 An implementation claiming to implement the General Profile for

Identifiers shall do so in accordance with the specifications in Section 3.1, General Security Profile for Identifiers.

Alternatively, it shall declare that it uses a modification, and provide a precise list of characters that are added to or removed from the profile.

C2 An implementation claiming to implement any of the following confusable-detection functions must do so in accordance with the specifications in Section 4, Confusable Detection.

1. X and Y are single-script confusables
2. X and Y are mixed-script confusables
3. X and Y are whole-script confusables
4. X has any simple single-script confusables
5. X has any mixed-script confusable
6. X has any whole-script confusable

Alternatively, it shall declare that it uses a modification, and provide a precise list of character mappings that are added to or removed from the provided ones.

C3 An implementation claiming to detect mixed scripts must do so in accordance with the specifications in Section 5.1, Mixed-Script Detection.

Alternatively, it shall declare that it uses a modification, and provide a precise specification of the differences in behavior.

C4 An implementation claiming to detect Restriction Levels must do so in accordance with the specifications in Section 5.2, Restriction-Level Detection.

Alternatively, it shall declare that it uses a modification, and provide a precise specification of the differences in behavior.

C5 An implementation claiming to detect mixed numbers must do so in accordance with the specifications in Section 5.3, Mixed-Number

Detection.

Alternatively, it shall declare that it uses a modification, and provide a precise specification of the differences in behavior.

Review Note: TODO add conformance clauses for 3.2 and 3.3.

3 Identifier Characters

Identifiers are special-purpose strings used for identification—strings that are deliberately limited to particular repertoires for that purpose. Exclusion of characters from identifiers does not affect the general use of those characters, such as within documents. Unicode Standard Annex #31, "Identifier and Pattern Syntax" [UAX31] provides a recommended method of determining which strings should qualify as identifiers. The UAX #31 specification extends the common practice of defining identifiers in terms of letters and numbers to the Unicode repertoire.

That specification also permits other protocols to use that method as a base, and to define a *profile* that adds or removes characters. For example, identifiers for specific programming languages typically add some characters like "$", and remove others like "-" (because of the use as *minus*), while IDNA removes "_" (among others)—see Unicode Technical Standard #46, "Unicode IDNA Compatibility Processing" [UTS46], as well as [IDNA2003], and [IDNA2008].

This document provides for additional identifier profiles for environments where security is an issue. These are profiles of the extended identifiers based on properties and specifications of the Unicode Standard [Unicode], including:

- The XID_Start and XID_Continue properties defined in the Unicode Character Database (see [DCore])
- The toCasefold(X) operation defined in *Chapter 3, Conformance* of [Unicode]
- The NFKC and NFKD normalizations defined in *Chapter 3, Conformance* of [Unicode]

The data files used in defining these profiles follow the UCD File Format, which has a semicolon-delimited list of data fields associated with given characters, with each field referenced by number. For more details, see [UCDFormat].

3.1 General Security Profile for Identifiers

The file [idmod] provides data for a profile of identifiers in environments where security is at issue. The file contains a set of characters recommended to be restricted from use. It also contains a small set of characters that are recommended as additions to the list of characters defined by the XID_Start and XID_Continue properties, because they may be used in identifiers in a broader context than programming identifiers.

The Restricted characters are characters not in common use, and are can be blocked to further reduce the possibilities for visual confusion. They include the following:

- characters not in modern use
- characters only used in specialized fields, such as liturgical characters, phonetic letters, and mathematical letter-like symbols
- characters in limited use by very small communities

The principle has been to be more conservative initially, allowing for the set to be modified in the future as requirements for characters are refined. For information on handling modifications over time, see *Section 2.9.1, Backward Compatibility* in *Unicode Technical Report #36, "Unicode Security Considerations"* [UTR36].

An implementation following the General Security Profile does not permit *Restricted* characters, unless it documents the additional characters that it does allow. Common candidates for such additions include characters for scripts listed in *Table 6, Aspirational Use Scripts* and *Table 7, Limited Use Scripts* of [UAX31]. However, characters from these scripts have not been a priority for examination for confusables or to determine specialized, non-modern, or uncommon-use characters.

Canonical equivalence is applied when testing candidate identifiers for inclusion of *Allowed* characters. For example, suppose the candidate string is the sequence

<u, *combining-diaeresis*>

The target string would be Allowed in *either* of the following 2 situations:

1. u is Allowed and ¨ is Allowed, or
2. ü is Allowed

In the file [idmod], Field 1 is the character in question, Field 2 is a **Status** value (either *Restricted* or *Allowed*), and Field 3 is a **Type** value. The **Type** values are selection factors used in deriving the **Status** value, and are listed in Table 1. Identifier Status and Type. They are included for use in customization.

Review Note: For the beta of v9.0, the data in Field 1 and Field 2 has been split into 2 different files, *IdentifierStatus* and *IdentifierType*.

The format for IdentifierStatus is the following. All characters not listed are IdentifierType=Restricted, so it only lists characters with IdentifierStatus=Allowed.

For example:

002D..002E ; Allowed # 1.1 HYPHEN-MINUS..FULL STOP

The format for IdentifierType is the following. The value is a space-delimited set (like Script Extensions), instead of only listing the "strongest" reason for exclusion. This allows the values to be used for more nuanced filtering. For example, if an implementation wants to allow an Exclusion script, it could still exclude Obsolete and Deprecated characters in that script. All characters not listed are IdentifierType={Recommended}.

For example:

2460..24EA ; Technical Not_XID Not_NFKC # 1.1 CIRCLED DIGIT ONE..CIRCLED DIGIT ZERO

We would like feedback as to whether this split into different files is useful or not. The original format xidmodifications.txt (with updated data) file is still included during the beta phase for comparison.

Table 1. Identifier Status and Type

Status	Type	Description
Restricted	Not_Character	Unassigned characters, private use characters, surrogates, most control characters
	Deprecated	Characters with the Unicode property Deprecated=Yes
	Default_Ignorable	Characters with the Unicode property Default_Ignorable_Code_Point=Yes
	Not_NFKC	Characters that cannot occur in strings normalized to NFKC.
	Not_XID	Other characters that do not qualify as default Unicode identifiers; that is, they do not have the Unicode property XID_Continue=True.
	Exclusion	Characters from scripts that are not in customary modern use: Table 4, Candidate Characters for Exclusion from Identifiers from [UAX31]
	Obsolete	Characters that are no longer in modern use.
	Technical	Specialized usage: technical, liturgical, etc.
	Uncommon_Use	Characters whose status is uncertain, or that are not commonly used in modern text.
	Limited_Use	Characters from scripts that are in limited use: Table 7, Limited Use Scripts in [UAX31].
	Aspirational	Characters from scripts would otherwise qualify as Limited Use, but have strong current efforts to increase their usage: Table 6, Aspirational Use Scripts in [UAX31].
Allowed	**Inclusion**	Exceptional allowed characters, including Table 3, Candidate Characters for Inclusion in Identifiers in [UAX31], and some characters for IDNA2008.

	Recommended	Table 5, Recommended Scripts in [UAX31]

For stability considerations, see Migrating Persistent Data.

The distinctions among the **Type** values is not strict; if there are multiple Types for restricting a character only one is given. The important characteristic is the **Status**: whether or not the character is Restricted. *As more information is gathered about characters, this data may change in successive versions.* That can cause either the **Status** or **Type** to change for a particular character. Thus users of this data should be prepared for changes in successive versions, such as by having a grandfathering policy in place for previously supported characters or registrations. Both **Status** and **Type** values are to be compared case-insensitively and ignoring hyphens and underbars.

Restricted characters should be treated with caution in registration, and disallowed unless there is good reason to allow them in the environment in question. However, the set of ***Status***=*Allowed* characters are not typically used as-is by implementations. Instead, they are applied as filters to the set of characters C that are supported by the identifier syntax, generating a new set C′. Typically there are also particular characters or classes of characters from C that are retained as **Exception** characters.

$$C' = (C \cap \{\textbf{\textit{Status}}=\textit{Allowed}\}) \cup \textbf{Exceptions}$$

The implementation may simply restrict use of new identifiers to C′, or may apply some other strategy. For example, there might be an appeal process for registrations of ids that contain characters outside of C' (but still inside of C), or in user interfaces for lookup of identifiers, warnings of some kind may be appropriate. For more information, see [UTR36].

The **Exception** characters would be implementation-specific. For example, a particular implementation might extend the default Unicode identifier syntax by adding **Exception** characters with the Unicode property *XID_Continue=False*, such as “$”, “-”, and “.”. Those characters are specific to that identifier syntax, and would be retained even though they are not in the ***Status***=*Allowed* set. Some implementations may also wish to add some [CLDR] exemplar characters for particular supported languages that have unusual characters.

The **Type**=*Inclusion* characters already contain some characters that are not letters or numbers, but that are used within words in some languages. For example, it is recommended that U+00B7 (·) MIDDLE DOT be allowed in identifiers, because it is required for Catalan.

The implementation may also apply other restrictions discussed in this document, such as checking for confusable characters or doing mixed-script detection.

3.2 IDN Security Profiles for Identifiers

Version 1 of this document defined operations and data that apply to [IDNA2003], which has been superseded by [IDNA2008] and Unicode Technical Standard #46, "Unicode IDNA Compatibility Processing" [UTS46]. The identifier modification data can be applied to whichever specification of IDNA is being used. For more information, see the [IDN

FAQ].

However, implementations can claim conformance to other features of this document as applied to domain names, such as Restriction Levels.

Review Notes:

- Should we add a section providing a way for implementations to specify restrictions on characters in the rest of the URL, aside from the domain name? That would include the path, query, or fragment.

3.3 Email Security Profiles for Identifiers

The *SMTP Extension for Internationalized Email* provides for specifications of internationalized email addresses [EAI]. However, it does not provide for testing those addresses for security issues. This section provides an email security profiles that may be used for that. It can be applied for different purposes, such as:

1. When an email address is registered, flag anything that doesn't meet the profile:
   - Either forbid the registration, or
   - Allow for an appeals process.
2. When an email address is detected in linkification of plain text:
   - Don't linkify if the identifier doesn't meet the profile.
3. When an email address is displayed in incoming email:
   - Flag it as suspicious with a wavy underline, if it doesn't meet the profile.
   - Filter characters from the quoted-string-part to prevent display problems.

This profile does not exclude characters from EAI. Instead, it provides a profile that can be used for registration, linkification, and notification. The goal is to flag "structurally unsound" and "unexpectedly garbagey" addresses.

Review Notes:

- Should we add another section or clauses about characters/sequences that really should always be completely excluded, meaning clients shouldn't ever recognize them as valid email addresses?
- Feedback on the exact conditions listed below is welcome.

An email address is formed from three main parts. (There are more elements of an email address, but these are the ones for which Unicode security is important.) For example:

```
"Joey" <joe31834@gmail.com>
```

- The **domain-part** is "gmail.com".
- The **local-part** is "joe31834"
- The **quoted-string-part** is "Joey"

To meet the requirements of the **Email Security Profiles for Identifiers** section of this specification, an identifier must satisfy the following conditions for the specified <restriction level>:

Domain-Part

The domain-part of an email address must satisfy Section 3.2 IDN Security Profiles for Identifiers, and satisfy the conformance clauses of [UTS46].

Local-Part

The local-part of an email address must satisfy all the following conditions:

1. It must be in NFKC format.
2. It must have level = <restriction level> or less, from Restriction_Level_Detection
3. It must not have mixed number systems according to Mixed_Number_Detection
4. It must satisfy *dot-atom-text* from RFC 5322 §3.2.3, where *atext* is extended as follows:

Where C ≤ U+007F, C is defined as in §3.2.3. (That is, C ∈ [!#-'*+\-/-9=?A-Z\^-~]. This list copies what is already in §3.2.3, and follows HTML5 for ASCII.)

Review note: It might be useful to limit the non-alphanums somewhat: they are rare in email, and can be confusing.

Where C > U+007F, both of the following conditions are true:

1. C has Status=Allowed from General_Security_Profile
   Review Note: We may drop some or all of the Type=Inclusion characters.
2. If C is the first character, it must be XID_Start from Default_Identifier_Syntax in [UAX31]
   Review Note: We could allow some extra symbols in 4.b.ii that are unproblematic, much as Swift Variables do.

Note that in RFC 5322 §3.2.3:

1. `dot-atom-text   =   1*atext *("." 1*atext)`
   - That is, dots can also occur in the local-part, but not leading, trailing, or two in a row.
   - In more conventional regex syntax, this would be:

2. `dot-atom-text   =   atext+ ("." atext+)*`

Note that BIDI controls and other format characters are specifically disallowed in the local-part, according to the above.

Quoted-String-Part

The quoted-string-part of an email address must satisfy the following conditions:

1. It must be in NFC.
   - Review note: we might loosen this to be NFKC, or add restrictions such as no default-ignorable characters (except for certain characters, like joiners).
2. It must not contain any stateful BIDI format characters.
   - That is, no [:bidicontrol:] except for the MARKs, since the bidi controls could influence the ordering of characters outside the quotes.
3. It must not contain more than four nonspacing marks in a row, and no sequence of two of the same nonspacing marks.
4. It may contain mixed scripts, symbols (including emoji), and so on.

Other Issues

The restrictions above are insufficient to prevent bidi-reordering that could intermix the quoted-string-part with the local-part or the domain-part in display. To prevent that, implementations could use bidi isolates (or equivalent) around the each of these parts.

Review Notes: Add that there are other possible issues, including confusability.

Review Notes: Add to UTR#36 more descriptions of email issues, and the use of RLO to mask dangerous file names, etc.

A serious practical issue is that clients do not know what the identity rules are for any particular email server: that is, when two email addresses are considered equivalent. For example, are *mark@macchiato.com* and *Mark@macchiato.com* the same by the server? Unfortunately, there is no way to query a server to see what identity rules it follows. One of the techniques used to deal with this problem is having whitelists of email providers indicating which of them are case-insensitive, dot-insensitive, or both.

4 Confusable Detection

The data in [confusables] provide a mechanism for determining when two strings are visually confusable. The data in these files may be refined and extended over time. For information on handling modifications over time, see *Section 2.9.1, Backward Compatibility* in Unicode Technical Report #36, "Unicode Security Considerations" [UTR36] and the Migration section of this document.

Collection of data for detecting gatekeeper-confusable strings is not currently a goal for the confusable detection mechanism in this document. For more information, see *Section 2 Visual Security Issues* in [UTR36].

The data provides a mapping from source characters to target strings.

To see whether two strings X and Y are confusable (abbreviated as X ≅ Y), an implementation uses a transform of X called a *skeleton(X)* defined by:

1. Converting X to NFD format, as described in [UAX15].
2. Successively mapping each source character in X to the target string according to the specified data.
3. Reapplying NFD.

The resulting strings *skeleton*(X) and *skeleton*(Y) are then compared. If they are identical (codepoint-for-codepoint), then X ≅ Y.

Note: The strings *skeleton*(X) and *skeleton*(Y) are ***not*** intended for display, storage or transmission. They should be thought of as an intermediate processing form, similar to a hashcode. The characters in *skeleton*(X) and *skeleton*(Y) are ***not*** guaranteed to be identifier characters.

Many of the processes in this document use the Script_Extensions (scx) property. When that property is used, its values are first (logically) transformed so that Inherited → Common, and certain script values are added:

- scx={...Han...} → {...Han, Jpan, Kore...}
- scx={...Hiragana...} → {...Hiragana, Jpan...}
- scx={...Katakana...} → {...Katakana, Jpan...}
- scx={...Hangul...} → {...Han, Kore...}

Review Note: Where the Script property is used below, it will be changed to use the Script_Extensions property, and the above text will be referenced.

Definitions

X and Y are *single-script confusables* if they are confusable, and each of them is a single script string according to *Section 5, Mixed-Script Detection, and it is the same script for each. Examples:* "so̷s" and "søs" in Latin, where the first word has the character "o" followed by the character U+0337 (̷) COMBINING SHORT SOLIDUS OVERLAY.

X and Y are *mixed-script confusables* if they are confusable but they are not single-script confusables. Examples: "paypal" and "pаypаl", where the second word has the character U+0430 (а) CYRILLIC SMALL LETTER A.

X and Y are *whole-script confusables* if they are *mixed-script confusables,* and each of them is a single script string*.* Example: "scope" in Latin and "ѕсоре" in Cyrillic.

Characters with the Script_Extension property values COMMON or INHERITED are ignored when testing for differences in script.

Each line in the data file has the following format: Field 1 is the source, Field 2 is the target, and Field 3 is obsolete . Field 3 used to contain different types, but now only has the value MA, which stands for "Mixed-Script, Any-Case".

For example:

0441 ; 0063 ; MA # (с → c) CYRILLIC SMALL LETTER ES → LATIN SMALL LETTER C #

2CA5 ; 0063 ; MA # (ⲥ → c) COPTIC SMALL LETTER SIMA → LATIN SMALL LETTER C # →ϲ→

Everything after the # is a comment and is purely informative. A asterisk after the comment indicates that the character is not an XID character [UAX31]. The comments provide the character names. If the data was derived via transitivity, there is an extra comment at the end. For instance, in the above example the derivation was:

1. ⲥ (U+2CA5 COPTIC SMALL LETTER SIMA)
2. → ϲ (U+03F2 GREEK LUNATE SIGMA SYMBOL)
3. → c (U+0063 LATIN SMALL LETTER C)

To reduce security risks, it is advised that identifiers use casefolded forms, thus eliminating uppercase variants where possible.

The data may change between versions. Even where the data is the same, the order of lines in the files may change between versions. For more information, see Migration.

Implementations that use the confusable data do not have to recursively apply the mappings, because the transforms are idempotent. That is,

skeleton(skeleton(X)) = skeleton(X)

- **Note:** due to production problems, versions before 7.0 did not maintain idempotency in all cases. For more information, see Migration.

This mechanism imposes transitivity on the data, so if X ≅ Y and Y ≅ Z, then X ≅ Z. It is possible to provide a more sophisticated confusable detection, by providing a metric between given characters, indicating their "closeness." However, that is computationally much more expensive, and requires more sophisticated data, so at this point in time the simpler mechanism has been chosen. That means that in some cases the test may be overly inclusive. However the frequency of such cases in real data should be small.

4.1 Whole-Script Confusables

This section specifies how test whether a string has a whole-script confusables, such as "scope" in Latin and "ѕсоре" in Cyrillic. The results depend on the set of characters that are accepted by the implementation.

The following gives the logical process for determing whether a single-script string *source string* has a whole-script confusable, given the implementation repertoire of characters R.

1. If the source string is mixed script, then return false. Otherwise transform the source string into nfd, called nfd-source.
2. Generate the set of all variants of nfd-source, using all of the combinations for each character from the equivalence classes in the confusables.txt file, filtered to keep only characters in R.
3. Remove all combinations that have mixed scripts, according to Mixed_Script_Detection, and remove the nfd-source string.
4. If that remainder set is not empty, then there is a whole-script confusable for the original.
5. If one of the remainder set has the same script from nfd-source, then there is a same-script confusable for the original.

Example:

1. The nfd-source for the source string is AB.
2. Assume A has the equivalence class {A, X, ZW}, and B has the equivalence class {B, C}. Then the result of #2 is {AB, AC, XB, XC, ZWB, ZWC}
3. Assume A is Latin, C and Z are Hiragana, and the others are Common. Then the remainders after removing mixed-script strings are: {XB, XC, ZWB, ZWC}
4. Because that set is not empty, there is a whole-script confusable for the input string.
5. For this example, there are none.

The logical description can be used for a reference implementation for testing, but is not particularly efficient. A production implementation can be optimized to incrementally test for mixed scripts as the combinations in step 2 are built up, and remove any initial substring that fails. That avoids adding the set tree of combinations that start with that initial substring without having to compute them in the first place. For example:

1. Process nfd-source character by character
2. Start with a mapping of scripts to samples, where each sample is initially “”.
3. Get each successive character’s confusable equivalence class as a set.
   a. Filter to remove entries with characters that are not in R. If the remaining set is empty, drop the script mapping
   b. For each script in the mapping, find a string in the remaining set that can be appended and yet remain in that script (avoiding the original character from nfd-source, where possible). If there is no such string, drop the script mapping
4. At the end of this process, drop any <script, nfd-source> entry.
5. The result is a mapping from the scripts to a sample whole-script confusable for the input in that script.

The above can be further optimized:

1. The mapping of characters to confusable equivalence classes can be preprocessed to filter out characters not in R, and filtered to remove strings with conflicting scripts. That makes step 3a faster.
2. A mapping can be produced that replaces each confusable equivalence class set by a map from script to characters. Note that the same string can appear under multiple scripts. That makes step 3b faster.
3. If the implementation does not require explicit string samples for the scripts, the algorithm can be recast to operate on sets of scripts instead. There is one complication to this: an entry that has only one character needs to be marked specially, so that it can be taken into account for step 4 above (removing generated strings that are identical to nfd-source).

Review Note: The former contents of this section will be removed.

~~Data is also provided for testing a string to see if a string X has any whole-script confusable, using the file [confusablesWS]. This file consists of a list of lines of the form:~~

```
~~<range>; <sourceScript>; <targetScript>; <type> #comment~~
```

~~The types are either L for lowercase-only, or A for any-case, where the any-case ranges are broader (including uppercase and lowercase characters). If the string is only lowercase, use the lowercase-only table. Otherwise, first test according to the any-case table, then casefold the string and test according to the lowercase-only table.~~

~~In using the data, all lines with the same *sourceScript* and *targetScript* are collected together to form a set of Unicode characters, after filtering to the **Allowed** characters from *Section 3.1, General Security Profile for Identifiers* . Logically, the file is a set of tuples of the form <*sourceScript, unicodeSet, targetScript*>. For example, the following lines are present for Latin to Cyrillic:~~

```
~~0061        ; Latn; Cyrl; L #     (a)     LATIN SMALL LETTER A~~
~~0063..0065  ; Latn; Cyrl; L # [3] (c..e)  LATIN SMALL LETTER C..LATIN SMALL LETTER E~~
~~...~~
~~0292        ; Latn; Cyrl; L #     (ʒ)     LATIN SMALL LETTER EZH~~
```

~~They logically form a tuple <*Latin, [a c-e ... \u0292], Cyrillic*>, which indicates that a Latin string containing characters only from that Unicode set can have a whole-script confusable in Cyrillic (lowercase-only). Note that if the implementation needs a set of **allowed** characters that is different from those in *Section 3.1, General Security Profile for Identifiers*, this process needs to be used to generate a different set of data.~~

~~Once the data is available, the following process is used:~~

1. ~~Convert the givenString to NFD format, as specified in [UAX15]~~
2. ~~Let *givenSet* be the set of all characters in givenString~~
3. ~~Remove all [:script=common:] and [:script=inherited:] characters from *givenSet*~~

4. ~~Let *givenScript* be the script of the characters in *givenSet*~~
    - ~~(if there is more than one script, fail with error).~~
5. ~~See if there is a tuple <*sourceScript, unicodeSet, targetScript*> where~~
    - ~~*sourceScript = givenScript*~~
    - ~~*unicodeSet* ⊇ givenSet~~
6. ~~If so, then there is a whole-script confusable in *targetScript*~~

~~The test is actually slightly broader than a whole-script confusable test. It tests whether the given string has a whole-script confusable string in another script, possibly with the addition or removal of common/inherited characters such as numbers and combining marks characters to both strings. In practice, however, this broadening has no significant impact.~~

~~Implementations would normally read the data into appropriate data structures in memory for processing. A quick additional optimization is to keep, for each script, a *fastReject* set, containing characters in the script contained in none of the *unicodeSet* values.~~

~~The following Java sample shows how this can be done (using the Java version of [ICU]):~~

```
~~/*~~
~~ * For this routine, we do not care what the target scripts are,~~
~~ * just whether there is at least one whole-script confusable.~~
~~ */~~
~~boolean hasWholeScriptConfusable(String s) {~~
~~        int givenScript = getSingleScript(s);~~
~~        if (givenScript == UScript.INVALID_CODE) {~~
~~                throw new IllegalArgumentException("Not single script string");~~
~~        }~~
~~        UnicodeSet givenSet = new UnicodeSet()~~
~~                .addAll(s)~~
~~                .removeAll(commonAndInherited);~~
~~        if (fastReject[givenScript].containsSome(givenSet)) return false;~~
~~        UnicodeSet[] possibles = scriptToUnicodeSets[givenScript];~~
~~        for (int i = 0; i < possibles.length; ++i) {~~
~~                if (possibles[i].containsAll(givenSet)) return true;~~
~~        }~~
~~        return false;~~
~~}~~
```

~~The data in [confusablesWS] is built using the data in [confusables], and subject to the same caveat: The data in these files may be refined and extended over time. For information on handling that, see *Section 2.9.1, Backward Compatibility* of [UTR36].~~

4.2 Mixed-Script Confusables

To test for mixed-script confusables, use the following process:

1. Convert the given string to NFD format, as specified in [UAX15].
2. For each script found in the given string, see if all the characters in the string outside of that script have whole-script confusables for that script (according to

Section 4.1, Whole-Script Confusables) .

Example 1: "paypal", with Cyrillic "a"s.

There are two scripts, Latin and Cyrillic. The set of Cyrillic characters {a} has a whole-script confusable in Latin. Thus the string is a mixed-script confusable.

Example 2: "toys-я-us", with one Cyrillic character "я".

The set of Cyrillic characters {я} does not have a whole-script confusable in Latin (there is no Latin character that looks like "я", nor does the set of Latin characters {o s t u y} have a whole-script confusable in Cyrillic (there is no Cyrillic character that looks like "t" or "u"). Thus this string is not a mixed-script confusable.

Example 3: "1ive", with a Greek "v" and Cyrillic "e".

There are three scripts, Latin, Greek, and Cyrillic. The set of Cyrillic characters {e} and the set of Greek characters {v} each have a whole-script confusable in Latin. Thus the string is a mixed-script confusable.

5 Detection Mechanisms

5.1 Mixed-Script Detection

The Unicode Standard supplies information that can be used for determining the script of characters and detecting mixed-script text. The determination of script is according to the Unicode Standard Annex #24, "Unicode Script Property" *[UAX24]* , using data from the Unicode Character Database [UCD]. For a given input string, the logical process is the following:

Define a set of sets of scripts SOSS.

For each character in the string:

1. Use the Script_Extensions property to find the set of scripts that the character has.
2. Remove Common and Inherited from that set of scripts.
3. If the result is not empty, add that set to SOSS.

If no single script is common to all of the sets in SOSS, then the string contains mixed scripts.

Characters with the script values *Common* and *Inherited* are ignored, because they are used with more than one script. For example, "abc-def" counts as a single script Latin because the script of "-" is ignored.

A set of scripts S is said to *cover* a SOSS *if* S intersects each element of SOSS. For example, {Latin, Greek} covers {{Latin, Georgian}, {Greek, Cyrillic}}, because:

1. {**Latin**, Greek} intersects {**Latin**, Georgian} (the intersection being {**Latin**}).

2. {Latin, **Greek**} intersects {**Greek**, Cyrillic} (the intersection being {**Greek**}).

The actual implementation of this algorithm can be optimized; as usual, the specification only depends on the results. The following Java sample using [ICU] shows how the above process can be implemented:

```
public static boolean isSingleScript(String identifier) {
    // Non-optimized code, for simplicity
    Set<BitSet> setOfScriptSets = new HashSet<BitSet>();
    BitSet temp = new BitSet();
    int cp;
    for (int i = 0; i < identifier.length(); i += Character.charCount(i)) {
        cp = Character.codePointAt(identifier, i);
        UScript.getScriptExtensions(cp, temp);
        if (temp.cardinality() == 0) {
            // HACK for older version of ICU
            final int script = UScript.getScript(cp);
            temp.set(script);
        }
        temp.andNot(COMMON_AND_INHERITED);
        if (temp.cardinality() != 0 && setOfScriptSets.add(temp)) {
            // If the set hasn't been added already,
            // add it and create new temporary for the next pass,
            // so we don't rewrite what's already in the set.
            temp = new BitSet();
        }
    }
    if (setOfScriptSets.size() == 0) {
        return true; // trivially true
    }
    temp.clear();
    // check to see that there is at least one script common to all the sets
    boolean first = true;
    for (BitSet other : setOfScriptSets) {
        if (first) {
            temp.or(other);
            first = false;
        } else {
            temp.and(other);
        }
    }
    return temp.cardinality() != 0;
}
```

This formulation ignores Common and Inherited scripts, and returns an error when a string contains mixed scripts.

5.2 Restriction-Level Detection

Restriction Levels 1-5 are defined here 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*. The lists of Recommended and Aspirational scripts are taken from *Table 5, Recommended Scripts* and *Table 6, Aspirational Use Scripts* of [UAX31]. For more information on the use of Restriction Levels, see *Section 2.9 Restriction Levels and Alerts* in [UTR36].

Whenever scripts are tested for in the following definitions, characters with Script_Extension=Common and Script_Extension=Inherited are ignored.

1. **ASCII-Only**
    - All characters in each identifier must be ASCII
2. **Single Script**
    - All characters in each identifier must be from a single script.
3. **Highly Restrictive**
    - All characters in each identifier must be from a single script, or from the combinations:
        - *Latin* + *Han* + *Hiragana* + *Katakana*;
        - *Latin* + *Han* + *Bopomofo*; or
        - *Latin* + *Han* + *Hangul*
    - No characters in the identifier can be outside of the Identifier Profile

    Note that this level will satisfy the vast majority of users.
4. **Moderately Restrictive**
    - Allow *Latin* with other Recommended or Aspirational scripts except *Cyrillic and Greek*
    - Otherwise, the same as **Highly Restrictive**
5. **Minimally Restrictive**
    - Allow arbitrary mixtures of scripts, such as Ωmega, Teχ, HλLF-LIFE, Toys-Я-Us.
    - Otherwise, the same as **Moderately Restrictive**
6. **Unrestricted**
    - Any valid identifiers, including characters outside of the Identifier Profile, such as I♥NY.org

These levels can be detected by reusing some of the mechanisms of Section 5.1. For a given input string, the Restriction Level is determined by the following logical process:

1. If the string contains any characters outside of the identifer profile, return **Unrestricted**.
2. If no character in the string is above 0x7F, return **ASCII**.
3. Compute SOSS as in Mixed Script Detection.
4. If a single script covers SOSS, return **Single Script**.
5. If any of the following sets cover SOSS, return **Highly Restrictive.**
    - {*Latin, Han, Hiragana, Katakana*}
    - {*Latin, Han, Bopomofo*}
    - {*Latin, Han, Hangul*}
6. Remove Latin from each element of SOSS. Then if SOSS contains any single **Recommended** or **Aspirational** script except *Cyrillic or Greek*, return **Moderately Restrictive**.
7. Otherwise, return **Minimally Restrictive**.

The actual implementation of this algorithm can be optimized; as usual, the specification only depends on the results.

5.3 Mixed-Number Detection

There are three different types of numbers in Unicode. Only numbers with General_Category = Decimal_Numbers (Nd) should be allowed in identifiers. However, characters from different decimal number systems can be easily confused. For example, U+0660 (٠) ARABIC-INDIC DIGIT ZERO can be confused with U+06F0 (۰) EXTENDED ARABIC-INDIC DIGIT ZERO, and U+09EA (৪) BENGALI DIGIT FOUR can be confused with U+0038 (8) DIGIT EIGHT.

For a given input string which does not contain non-decimal numbers, the logical process of detecting mixed numbers is the following:

For each character in the string:

1. Find the decimal number value for that character, if any.
2. Map the value to the unique zero character for that number system.

If there is more than one such zero character, then the string contains multiple decimal number systems.

The actual implementation of this algorithm can be optimized; as usual, the specification only depends on the results. The following Java sample using [ICU] shows how this can be done :

```
    public UnicodeSet getNumberRepresentatives(String identifier) {
        int cp;
        UnicodeSet numerics = new UnicodeSet();
        for (int i = 0; i < identifier.length(); i += Character.charCount(i)) {
            cp = Character.codePointAt(identifier, i);
            // Store a representative character for each kind of decimal digit
            switch (UCharacter.getType(cp)) {
            case UCharacterCategory.DECIMAL_DIGIT_NUMBER:
                // Just store the zero character as a representative for comparison.
                // Unicode guarantees it is cp - value.
                numerics.add(cp - UCharacter.getNumericValue(cp));
                break;
            case UCharacterCategory.OTHER_NUMBER:
            case UCharacterCategory.LETTER_NUMBER:
                throw new IllegalArgumentException("Should not be in identifiers.");
            }
        }
        return numerics;
    }
...
    UnicodeSet numerics = getMixedNumbers(String identifier);
    if (numerics.size() > 1) reject(identifer, numerics);
```

5.4 Optional Detection

There are additional enhancements that may be useful in spoof detection. This includes such mechanisms as marking strings as "mixed script" where they contain both simplified-only and traditional-only Chinese characters, using the Unihan data in the Unicode Character Database [UCD], or detecting sequences of the same nonspacing mark.

Other enhancements useful in spoof detection include the following:

1. Mark Chinese strings as "mixed script" if they contain both simplified (S) and traditional (T) Chinese characters, using the Unihan data in the Unicode Character Database [UCD].
    a. The criterion can only be applied if the language of the string is known to be Chinese. So, for example, the string “写真だけの結婚式 ” is Japanese, and should not be marked as mixed script because of a mixture of S and T characters.
    b. Testing for whether a character is S or T needs to be based not on whether the character *has* a S or T variant , but whether the character *is* an S or T variant.
2. Forbid sequences of the same nonspacing mark
3. Check to see that all the characters are in the sets of exemplar characters for at least one language in the Unicode Common Locale Data Repository [CLDR].

6 Development Process

As discussed in Unicode Technical Report #36, "Unicode Security Considerations" [UTR36], confusability among characters cannot be an exact science. There are many factors that make confusability a matter of degree:

- Shapes of characters vary greatly among fonts used to represent them. The Unicode Standard uses representative glyphs in the code charts, but font designers are free to create their own glyphs. Because fonts can easily be created using an arbitrary glyph to represent any Unicode code point, character confusability with arbitrary fonts can never be avoided. For example, one could design a font where the ‘a’ looks like a ‘b’ , ‘c’ like a ‘d’, and so on.
- Writing systems using contextual shaping (such as Arabic, and many South Asian systems) introduce even more variation in text rendering. Characters do not really have an abstract shape in isolation and are only rendered as part of cluster of characters making words, expressions, and sentences. It is a fairly common occurrence to find the same visual text representation corresponding to very different logical words that can only be recognized by context, if at all.
- Font style variants such as italics may introduce a confusability which does not exist in another style. For example, in the Cyrillic script, the U+0442 (т) CYRILLIC SMALL LETTER TE looks like a small caps Latin ‘T’ in normal style, while it looks like a small Latin ‘m’ in italic style.

In-script confusability is extremely user-dependent. For example, in the Latin script, characters with accents or appendices may look similar to the unadorned characters for some users, especially if they are not familiar with their meaning in a particular language. However, most users will have at least a minimum understanding of the range of characters in their own script, and there are separate mechanisms available to deal with other scripts, as discussed in [UTR36].

As described elsewhere, there are cases where the confusable data may be different than expected. Sometimes this is because two characters or two strings may only be confusable in some fonts. In other cases, it is because of transitivity. For example, the dotless and dotted I are considered equivalent (ı ↔ i), because they look the same when accents such as an *acute* are applied to each. However, for practical implementation usage, transitivity is sufficiently important that some oddities are accepted.

The data may be enhanced in future versions of this specification. For information on handling changes in data over time, see *Section 2.9.1, Backward Compatibility* of [UTR36].

6.1 Confusables Data Collection

The confusability data was created by collecting a number of prospective confusables, examining those confusables according to a set of common fonts, and processing the result for transitive closure.

The primary goal is to include characters that would be **Status**=*Allowed* as in *Table 1. Identifier Status and Type*. Other characters, such as NFKC variants, are not a primary focus for data collection. However, such variants may certainly be included in the data, and may be submitted using the online forms at [Feedback].

The prospective confusables were gathered from a number of sources. Erik van der Poel contributed a list derived from running a program over a large number of fonts to catch characters that shared identical glyphs within a font, and Mark Davis did the same more recently for fonts on Windows and the Macintosh. Volunteers from Google, IBM, Microsoft and other companies gathered other lists of characters. These included native speakers for languages with different writing systems. The Unicode compatibility mappings were also used as a source. The process of gathering visual confusables is ongoing: the Unicode Consortium welcomes submission of additional mappings. The complex scripts of South and Southeast Asia need special attention. The focus is on characters that can be in the Recommended profile for identifiers, because they are of most concern.

The fonts used to assess the confusables included those used by the major operating systems in user interfaces. In addition, the representative glyphs used in the Unicode Standard were also considered. Fonts used for the user interface in operating systems are an important source, because they are the ones that will usually be seen by users in circumstances where confusability is important, such such as when using IRIS (Internationalized Resource Identifiers) and their sub-elements (such as domain names). These fonts have a number of other relevant characteristics:

- They rarely changed in updates to operating systems and applications; changes brought by system upgrades tend to be gradual to avoid usability disruption.
- Because user interface elements need to be legible at low screen resolution (implying a low number of pixels per EM), fonts used in these contexts tend to be designed in sans-serif style, which has the tendency to increase the possibility of confusables. There are, however, some languages such as Chinese where a serif style is in common use.
- Strict bounding box requirements create even more constraints for scripts which use relatively large ascenders and descenders. This also limits space allocated for accent or tone marks, and can also create more opportunities for confusability.

Pairs of prospective confusables were removed if they were always visually distinct at common sizes, both within and across fonts. The data was then closed under transitivity, so that if X≅Y and Y≅Z, then X≅Z. In addition, the data was closed under substring operations, so that if X≅Y then AXB≅AYB. It was then processed to produce the in-script and cross-script data, so that a single data table can be used to map an input string to a resulting *skeleton*.

A skeleton is intended *only* for internal use for testing confusability of strings; the resulting text is not suitable for display to users, because it will appear to be a hodgepodge of different scripts. In particular, the result of mapping an identifier will not necessary be an identifier. Thus the confusability mappings can be used to test whether two identifiers are confusable (if their skeletons are the same), but should definitely not be used as a "normalization" of identifiers.

6.2 Identifier Modification Data Collection

The **idmod** data is gathered in the following way. The basic assignments are derived based on UCD character properties, information in [UAX31], and a curated list of exceptions based on information from various sources, including the core specification of the Unicode Standard, annotations in the code charts, information regarding CLDR exemplar characters, and external feedback.

The first condition that matches in the order of the items from top to bottom in Table 1. Identifier Status and Type is used, with a few exceptions:

1. When a character is in *Table 3, Candidate Characters for Inclusion in Identifiers* in [UAX31], then it is given the Type Inclusion, regardless of other properties.
2. When the Script_Extensions property value for a character contains multiple Script property values, the Script used for the derivation is the first in the following list:
    1. *Table 5, Recommended Scripts*
    2. *Table 6, Aspirational Use Scripts*
    3. *Table 7, Limited Use Scripts*
    4. *Table 4, Candidate Characters for Exclusion from Identifiers*
        - *Table 4* also has some conditions that are not dependent on script; those conditions are applied regardless of Script_Extensions property value.

The script information in *Table 4*, *Table 5*, *Table 6* and *Table 7* are in machine-readable form in CLDR, as scriptMetadata.txt.

7 Data Files

The following files provide data used to implement the recommendations in this document. The data may be refined in future versions of this specification. For more information, see *Section 2.9.1, Backward Compatibility* of [UTR36].

The Unicode Consortium welcomes feedback on additional confusables or identifier restrictions. There are online forms at [Feedback] where you can suggest additional characters or corrections.

The files are in http://www.unicode.org/Public/security/. The directories there contain data files associated with a given version. The directory for *this* version is:

http://www.unicode.org/Public/security/9.0.0

The data files for the latest approved version are also in the directory:

http://www.unicode.org/Public/security/latest

Data File List

[idmod]	xidmodifications.txt	**Identifier Modifications**: Provides the list of additions and restrictions recommended for building a profile of identifiers for environments where security is at issue. Review Note: this is intended to be replaced by two files, IdentifierStatus.txt and IdentifierType.txt. See 3.1 General Security Profile for Identifiers for details.
[confusables]	confusables.txt	**Visually Confusable Characters**: Provides a mapping for visual confusables for use in detecting possible security problems. The usage of the file is described in Section 4, Confusable Detection.
[confusablesSummary]	confusablesSummary.txt	**A summary view of the confusables**: Groups each set of confusables together, listing them first on a line starting with #, then individually with

		names and code points. See Section 4, Confusable Detection
~~[confusablesWS]~~	~~confusablesWholeScript.txt~~	~~**Whole Script Confusables:** Data for testing for the possible existence of whole-script and mixed-script confusables. See Section 4, Confusable Detection~~
[intentional]	intentional.txt	**Intentional Confusable Mappings:** A selection of characters whose glyphs in any particular typeface would probably be designed to be identical in shape when using a harmonized typeface design.

Review Note: The confusability data is subject to a number of constraints, as described in feedback to PRI #292. We'd be interested in proposals for addressing those constraints, while allowing effective collection of data.

1. Inclusion: X cannot be confusable with ...X...
   - This is probably the easiest constraint to loosen.
2. Transitivity: X ~ Y and Y ~ Z then X ~ Z
   - This would be more difficult, since it would call for a 'distance' metric between two characters (or strings), which would fundamentally change the data, how the data is collected, and the algorithm implementers would use
3. Context: X ~ Y if and only if XZ ~ YZ
   - This would allow for contextual forms to make a difference. For a constrained set of characters this might be doable, but we'd need a format for expressing the context; perhaps leveraging the format used in CLDR transforms.

Migration

Beginning with version 6.3.0, the version numbering of this document has been changed to indicate the version of the UCD that the data is based on. For versions up to and including 6.3.0, the following table shows the correspondence between the versions of this document and UCD versions that they were based on.

Version Correspondence

Version	Release Date	Data File Directory	UCD Version	UCD Date
Version 1	2006-08-15	/Public/security/revision-02/	5.1.0	2008-04
draft only	2006-08-11	/Public/security/revision-03/	n/a	n/a
Version 2	2010-08-05	/Public/security/revision-04/	6.0.0	2010-10
Version 3	2012-07-23	/Public/security/revision-05/	6.1.0	2012-01
6.3.0	2013-11-11	/Public/security/6.3.0/	6.3.0	2013-09

If an update version of this standard is required between the associated UCD versions, the version numbering will include an update number in the 3rd field. For example, if a version of this document and its associated data is needed between UCD 6.3.0 and UCD 7.0.0, then a version 6.3.**1** could be used.

Migrating Persistent Data

Implementations must migrate their persistent data stores (such as database indexes) whenever those implementations update to use the data files from a new version of this specification.

Stability is never guaranteed between versions, although it is maintained where feasible. In particular, an updated version of confusable mapping data may use a mapping for a particular character that is different from the mapping used for that character in an earlier version. Thus there may be cases where X → Y in Version N, and X → Z in Version N+1, where Z may or may not have mapped to Y in Version N. Even in cases where the logical data has not changed between versions, the order of lines in the data files may have been changed.

The Identifier Status does not have stability guarantees (such as "Once a character is Allowed, it will not become Restricted in future versions"), because the data is changing over time as we find out more about character usage. Certain of the Type values are backward compatible, such as Not_XID, but most may change as new data becomes available. The identifier data may also not appear to be completely consistent when just viewed from the perspective of script and general category. For example, it may well be that one character out of a set of non-spacing marks in a script is Restricted, while others are not. But that can be just a reflection of the fact that that character is obsolete and the others are not.

For identifier lookup, the data is aimed more at flagging possibly questionable characters, thus serving as one factor (among perhaps many, like using the "Safe Browsing" service) in determining whether the user should be notified in some way. For registration, flagged characters can result in a "soft no", that is, require the user to appeal a denial with more information.

For dealing with characters whose status changes to Restricted, implementations can use a grandfathering mechanism if they want to maintain backwards compatibility.

Implementations should therefore have a strategy for migrating their persistent data stores (such as database indexes) that use any of the confusable mapping data or other data files.

Version 8.0 Migration

In Version 8.0, the following changes were made to the Identifier Status and Type:

- Changed to the standard UCD formatting. For example, *limited-use* → *Limited_Use*.
    - Usually this was simply changing the case and hyphen, but *not-chars* changed to *Not_Character*.
- Aligned the Identifier Type better with UAX 31 and Unicode properties
    - historic
        - → Exclusion, where from *Table 4, Candidate Characters for Exclusion from Identifiers*,
        - → Obsolete, otherwise
    - limited-use
        - → Limited_Use, where from *Table 7, Limited Use Scripts*,
        - → Aspirational, where from *Table 6, Aspirational Use Scripts*
        - → Uncommon-Use, otherwise
    - obsolete
        - → Deprecated, where matching the Unicode property

Version 7.0 Migration

Due to production problems, versions of the confusable mapping tables before 7.0 did not maintain idempotency in all cases, so updating to version 8.0 is strongly advised.

Anyone using the skeleton mappings needs to rebuild any persistent uses of skeletons, such as in database indexes.

The SL, SA, and ML mappings in 7.0 were significantly changed to address the idempotency problem. However, the tables SL, SA, and ML were still problematic, and discouraged from use in 7.0. They were thus removed from version 8.0.

All of the data necessary for an implementation to recreate the removed tables is available in the remaining data (MA) plus the Unicode Character Database properties (script, casing, etc.). Such a recreation would examine each of the equivalence classes from the MA data, and filter out instances that didn't fit the constraints (of script or casing). For the target character, it would choose the most neutral character, typically a symbol. However, the reasons for deprecating them still stand, so it is not recommended that implementations recreate them.

Note also that as the Script_Extensions data is made more complete, it may cause characters in the whole-script confusables data file to no longer match. For more information, see *Section 4* *Confusable Detection*.

Acknowledgments

Mark Davis and Michel Suignard authored the bulk of the text, under direction from the Unicode Technical Committee. Steven Loomis and other people on the ICU team were very helpful in developing the original proposal for this technical report. Thanks also to the following people for their feedback or contributions to this document or earlier versions of it, or to the source data for confusables or idmod: Julie Allen, Andrew Arnold, Vernon Cole, David Corbett (specal thanks for the many contributions), Douglas Davidson, Rob Dawson, Alex Dejarnatt, Chris Fynn, Martin Dürst, Asmus Freytag, Deborah Goldsmith, Paul Hoffman, Denis Jacquerye, Cibu Johny, Patrick L. Jones, Peter Karlsson, Mike Kaplinskiy, Gervase Markham, Eric Muller, David Patterson, Erik van der Poel, Roozbeh Pournader, Michael van Riper, Marcos Sanz, Alexander Savenkov, Dominikus Scherkl, Manuel Strehl, Chris Weber, Kenneth Whistler, and Waïl Yahyaoui. Thanks to Peter Peng for his assistance with font confusables.

References

[CLDR]	Unicode Locales Project (Unicode Common Locale Data Repository) http://www.unicode.org/cldr/
[DCore]	Derived Core Properties http://www.unicode.org/Public/UNIDATA/DerivedCoreProperties.txt
[DemoConf]	http://unicode.org/cldr/utility/confusables.jsp
[DemoIDN]	http://unicode.org/cldr/utility/idna.jsp
[DemoIDNChars]	http://unicode.org/cldr/utility/list-unicodeset.jsp?a=\p{age%3D3.2}-\p{cn}-\p{cs}-\p{co}&abb=on&uts46+idna+idna2008
[FAQSec]	Unicode FAQ on Security Issues http://www.unicode.org/faq/security.html
[ICANN]	ICANN Documents: Internationalized Domain Names http://www.icann.org/en/topics/idn/ The IDN Variant Issues Project http://www.icann.org/en/topics/new-gtlds/idn-vip-integrated-issues-23dec11-en.pdf Maximal Starting Repertoire Version 2 (MSR-2) https://www.icann.org/news/announcement-

2-2015-04-27-en

[ICU] International Components for Unicode
http://site.icu-project.org/

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

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

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

- Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework
http://tools.ietf.org/html/rfc5890
- Internationalized Domain Names in Applications (IDNA) Protocol
http://tools.ietf.org/html/rfc5891
- The Unicode Code Points and Internationalized Domain Names for Applications (IDNA)
http://tools.ietf.org/html/rfc5892
- Right-to-Left Scripts for Internationalized Domain Names for Applications (IDNA)
http://tools.ietf.org/html/rfc5893

There are also informative documents:

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

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

[EAI]	https://tools.ietf.org/html/rfc6531
[Feedback]	To suggest additions or changes to confusables or identifier restriction data, please see: http://unicode.org/reports/tr39/suggestions.html For issues in the text, please see: Reporting Errors and Requesting Information Online http://www.unicode.org/reporting.html
[Reports]	Unicode Technical Reports http://www.unicode.org/reports/ For information on the status and development process for technical reports, and for a list of technical reports.
[RFC3454]	P. Hoffman, M. Blanchet. "Preparation of Internationalized Strings ("stringprep")", RFC 3454, December 2002. http://ietf.org/rfc/rfc3454.txt
[RFC3490]	Faltstrom, P., Hoffman, P. and A. Costello, "Internationalizing Domain Names in Applications (IDNA)", RFC 3490, March 2003. http://ietf.org/rfc/rfc3490.txt
[RFC3491]	Hoffman, P. and M. Blanchet, "Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)", RFC 3491, March 2003. http://ietf.org/rfc/rfc3491.txt
[RFC3492]	Costello, A., "Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)", RFC 3492, March 2003. http://ietf.org/rfc/rfc3492.txt
[Security-FAQ]	http://www.unicode.org/faq/security.html
[UCD]	Unicode Character Database. http://www.unicode.org/ucd/ For an overview of the Unicode Character Database and a list of its associated files.
[UCDFormat]	UCD File Format http://www.unicode.org/reports/tr44/#Format_Conventions

[UAX15]	UAX #15: Unicode Normalization Forms http://www.unicode.org/reports/tr15/
[UAX24]	UAX #24: Unicode Script Property http://www.unicode.org/reports/tr24/
[UAX29]	UAX #29: Unicode Text Segmentation http://www.unicode.org/reports/tr29/
[UAX31]	UAX #31: Unicode Identifier and Pattern Syntax http://www.unicode.org/reports/tr31/
[Unicode]	The Unicode Standard For the latest version, see: http://www.unicode.org/versions/latest/
[UTR36]	UTR #36: Unicode Security Considerations http://www.unicode.org/reports/tr36/
[UTS18]	UTS #18: Unicode Regular Expressions http://www.unicode.org/reports/tr18/
[UTS39]	UTS #39: Unicode Security Mechanisms http://www.unicode.org/reports/tr39/
[UTS46]	Unicode IDNA Compatibility Processing http://www.unicode.org/reports/tr46/
[Versions]	Versions of the Unicode Standard http://www.unicode.org/standard/versions/ For information on version numbering, and citing and referencing the Unicode Standard, the Unicode Character Database, and Unicode Technical Reports.

Modifications

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

Revision 12

- **Proposed Update** for Unicode 9.0.0.
- *Section 3.1 General Security Profile for Identifiers*
  - Added review note describing the planned split of the xidmodification.txt file into two files.

Section 3.2 IDN Security Profiles for Identifiers
  - Added clarification and review notes
- *Section 3.3 Email Security Profiles for Identifiers*
  - New section
- *Section 4 Confusable Detection*
  - Added text about the use of Script_Extensions, with a review note indicating remaining work to integrate that.
- *Section 4.1 Whole-Script Confusables*
  - Withdrew the **confusablesWholeScript.txt** data file, because in practice the process of derivation depends on the set of characters supported by the application. Instead, this section now describes the logical process of deriving the whole-script data based on the supported characters.
- Section 5.1 Mixed-Script Detection
  - Fixed typo in sample code
- *Section 7 Data Files*
  - Added review note soliciting feedback on how to improve the data represention to handle cases that can't currently be handled.
  - Added a review note about the split in the data files planned for v9.0.
  - Removed the reference to **confusablesWholeScript.txt**.

Previous revisions can be accessed with the "Previous Version" link in the header.
