Proposed Update Unicode Technical Standard #39

UNICODE SECURITY MECHANISMS

Version	3 draft 2
Editors	Mark Davis (markdavis@google.com), Michel Suignard (michel@suignard.com)
Date	2012-03-02
This Version	http://www.unicode.org/reports/tr39/tr39-5.html
Previous Version	http://www.unicode.org/reports/tr39/tr39-4.html
Latest Version	http://www.unicode.org/reports/tr39/
Latest Proposed Update	http://www.unicode.org/reports/tr39/proposed.html
Revision	5

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

This is a draft document which may be updated, replaced, or superseded by other documents at any time.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 Standard (UTS) is an independent specification.

Conformance to the Unicode Standard does not imply conformance to any UTS.

Please submit corrigenda and other comments with the online reporting form [Feedback]. Related information that is useful in understanding this document is found in References. For the latest version of the Unicode Standard see [Unicode]. For a list of current Unicode Technical Reports see [Reports]. For more information about versions of the Unicode Standard, see [Versions].

Contents

1 Introduction
2 Conformance
3 Identifier Characters
    3.1 General Security Profile for Identifiers
    3.2 IDN Security Profiles for Identifiers
4 Confusable Detection
    4.1 Whole-Script Confusables
    4.2 Mixed-Script Confusables
5 Detection Mechanisms
    5.1 Mixed-Script Detection
    5.2 Restriction-Level Detection
    5.3 Mixed-Number Detection
    5.4 Optional Detection
6 Development Process
    6.1 Data Collection
7 Data Files
Acknowledgements
References
Modifications

---

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.

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 removed 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 examined for confusables or to determine specialized, non-modern, or limited-use characters.

[Review Note: It would be easier for implementations to include the Aspirational-Use and Limited-Use scripts if there were separate types for limited-use-scripts and aspirational-scripts in Table 1. Feedback on this issue is welcome.]

In the file [idmod], Field 1 is the character in question, Field 2 is a **Status** (either restricted or allowed), and Field 3 is a **Type**. The Types are listed in Table 1, Identifier Modification Key:

Table 1. Identifier Modification Key

Status	Type	Description
restricted	default-ignorable	Characters with the Unicode property Default_Ignorable_Code_Point=True
restricted	historic	Characters not in customary modern use; includes Table 4, Candidate Characters for Exclusion from Identifiers from [UAX31]
restricted	limited-use	Characters whose status is uncertain, or that are used in limited environments, or those in Table 7, Limited Use Scripts and Table 6, Aspirational Use Scripts in [UAX31]
restricted	not-chars	Unassigned characters, private use characters, surrogates, most control characters
restricted	not-NFKC	Characters that cannot occur in strings normalized to NFKC.
restricted	not-xid	Other characters that do not qualify as default Unicode identifiers; that is, they do not have the Unicode property XID_Continue=True.
restricted	obsolete	Technical characters that are no longer in use; characters with the Unicode property Deprecated=True
restricted	technical	Technical characters
allowed	**inclusion**	Table 3, Candidate Characters for Inclusion in Identifiers in [UAX31]. See also the notes on MidLetter in [UAX29].
allowed	**recommended**	Table 5, Recommended Scripts in [UAX31] ~~(excluding restricted)~~

Restricted characters should be treated with caution in registration, and

disallowed unless there is good reason to allow them in the enviroment in question. In user interfaces for lookup of identifiers, warnings of some kind may be appropriate. For more information, see [UTR36].

When the sets of restricted and allowed characters are applied to a particular identifier syntax, modifications need to be made. For example, a particular syntax might extend the default Unicode identifier syntax by adding characters with the Unicode property XID_Continue=False, such as "$", "-", and ".". Those characters are specific to that identifier syntax, and would need to be retained even though they are not in the allowed set.

The distinctions among the Types 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 registrations.

This list is also used in deriving the IDN Identifiers list given below. It is, however, designed to be applied to general environments.

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 identifer modification data can be applied to whichever specification of IDNA is being used. For more information, see the [IDN FAQ].

4 Confusable Detection

The tables in the data file [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].

The data is organized into four different tables, depending on the desired parameters. Each table provides a mapping from source characters to target

strings. On the basis of this data, there are three main classes of confusable strings:

> X and Y are single-script confusables if they are confusable according to the Single-Script table, and each of them is a single script string according to Section 5, Mixed-Script Detection. Examples: "søs" 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 according to the Mixed-Script table, and they are not single-script confusables. Examples: "paypal" and "paypal", 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 "scope" in Cyrillic.

To see whether two strings X and Y are confusable according to a given table (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 table.
3. Reapplying NFD.

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

> **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.

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

$$\text{skeleton}(\text{skeleton}(X)) = \text{skeleton}(X)$$

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.

Each line in the data file has the following format: Field 1 is the source, Field 2 is the target, and Field 3 is a type identifying the table. For example,

309C ; 030A ; SL #* (゛ →゚) KATAKANA-HIRAGANA SEMI-VOICED SOUND MARK → COMBINING RING ABOVE # →゚→→ ゚→

The types are explained in Table 2, Confusable Data Table Types. 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:

- U+309A (゚) COMBINING KATAKANA-HIRAGANA SEMI-VOICED SOUND MARK →
- U+FF9F (ﾟ) HALFWIDTH KATAKANA SEMI-VOICED SOUND MARK →
- U+309C (゛) KATAKANA-HIRAGANA SEMI-VOICED SOUND MARK →
- U+030A (̊) COMBINING RING ABOVE

To reduce security risks, it is advised that identifiers use casefolded forms, thus eliminating uppercase variants where possible. Characters with the script values COMMON or INHERITED are ignored when testing for differences in script.

Table 2. Confusable Data Table Types

Type	Name	Description
SL	Single-Script, Lowercase	This table is used to test cases of single-script confusables, where both the source character and the target string are case folded. For example: # (ø → ø) LATIN SMALL LETTER O WITH STROKE → LATIN SMALL LETTER O, COMBINING SHORT SOLIDUS OVERLAY

SA	Single-Script, Any-Case	This table is used to test cases of single-script confusables, where the output allows for mixed case (which may be later folded away). For example, this table contains the following entry not found in SL: # (O → 0) LATIN CAPITAL LETTER O → DIGIT ZERO
ML	Mixed-Script, Lowercase	This table is used to test cases of mixed-script and whole-script confusables, where both the source character and the target string are case folded. For example, this table contains the following entry not found in SL or SA: # (ν → v) GREEK SMALL LETTER NU → LATIN SMALL LETTER V
MA	Mixed-Script, Any-Case	This table is used to test cases of mixed-script and whole-script confusables, where the output allows for mixed case (which may be later folded away). For example, this table contains the following entry not found in SL, SA, or ML: # (Ι → I) GREEK CAPITAL LETTER IOTA → LATIN SMALL LETTER L

4.1 Whole-Script Confusables

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.

To test whether a single-script string givenString has a whole-script confusable in targetScript, 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 {а} 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 ~~Mixed-Script~~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 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 isMultiScript(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].

[Review Note: the following text was moved from UTR36. The text was left unmarked except for the changes from the previous version of UTR36.]

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:
      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 ~~Latin-script~~ users.
3. **Moderately Restrictive**
    - Allow Latin with other Recommended or Aspirational scripts except Cyrillic and Greek~~, Cherokee~~
    - Otherwise, the same as **Highly Restrictive**
4. **Minimally Restrictive**
    - Allow arbitrary mixtures of scripts, such as 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, 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 **Highly_Restrictive**.

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:

- mark 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]
- forbid sequences of the same nonspacing mark
- 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 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.

6.1 Data Collection

The confusability tables were 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 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 tables, so that a single 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.

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].

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/beta

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

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

[zippedData]	uts39-data-4.0.zip	**A zipped version of all the data files.** [Review Note: this will be linked once the release is final.]

[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.
[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.

Acknowledgements

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: Julie Allen, Andrew Arnold, Douglas Davidson, Chris Fynn, Martin Dürst, Asmus Freytag, Deborah Goldsmith, Paul Hoffman, Denis Jacquerye, Cibu Johny, Patrick L. Jones, Peter Karlsson, Gervase Markham, Eric Muller, Erik van der Poel, Michael van Riper, Marcos Sanz, Alexander Savenkov, Dominikus Scherkl, Chris Weber, and Kenneth Whistler. 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

[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 ~~draft~~ 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

[Feedback] To suggest additions or changes to confusables or identifier restriction data, please see:
http://unicode.org/reports/tr39/suggestions.html

During the beta period, please submit other feedback at:
http://www.unicode.org/review/pri209/
[Review note: delete the above once out of the beta period.]

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/
For the 6.1 version, see:
http://www.unicode.org/versions/Unicode6.1.0/

[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 5

- Proposed update.
- Fixed reported typos, and updated references.
- Incorporated script extensions into mixed-script detection in Section 5.1, Mixed-Script Detection.
- Moved the definition of Restriction Level from UTR #36 into Section 5.2, Restriction-Level Detection.
- Explicitly defined the process of Restriction Level and Mixed Number detection (formerly discussed in general terms), and provided conformance clauses. Section 5.2, Restriction-Level Detection and Section 5.3, Mixed-Number Detection.
- Moved remaining items discussed in general terms into 5.4, Optional Detection.
- Updated table references to UAX #31 in Section 3.1, General Security Profile for Identifiers.

Revision 4

- Removed the idnchars.txt data file, Section 3.2, IDN Security Profiles for Identifiers, the old conformance clause C1; and renumbered previous C0 as C1.
- Moved the subsection Data Files to be Section 7.
- Added Table 1, Identifier_Modification_Key and text following, explaining the identifer restrictions. Especially see the caveat about use of the data.
- Added link to form for for submitting suggested data.
- Changed to NFD instead of NFKD, with relevant mappings moved into the data file.
- Revised the confusable data to add data extracted from a comparison of font data from windows and mac.
    - Data was generated for characters sharing the same outline in some font on that system.

    - Those were then reviewed to remove errors due to bad font mappings.
    - Additional mappings were also added, such as "rn"≅"m".
- The recommended characters in identifiers were updated based on UAX 31, with the following labels:
    - UAX #31 Table 4, Candidate Exclusions
    - UAX #31 Table 5, Limited Use
- The IICore information was removed, because it is not a good guide to usage.

Revision 3 being a proposed update, only changes between revisions 2 and 4 are noted here.

Revision 2

- Removed the "input" and "lenient" tables
- Minor editing and clarifications

Revision 1

- Created from Appendix A, B, and D from [UTR36].
- Created Section 6, Development Process based on document L2/06-055.
- Removed DITTO Mark, added intentional mappings
- Added 5.0 scripts to removals: Balinese, Cuneiform, Phoenician, Phags_Pa
- Revised table formats
- Added the intentional mappings, plus a pointer to source data

---