Internet Architecture Board (IAB)
Request for Comments: 6055
Updates: 2130
Category: Informational
ISSN: 2070-1721

D. Thaler
Microsoft
J. Klensin

S. Cheshire
Apple

February 2011

IAB Thoughts on Encodings for Internationalized Domain Names

Abstract

This document explores issues with Internationalized Domain Names (IDNs) that result from the use of various encoding schemes such as UTF-8 and the ASCII-Compatible Encoding produced by the Punycode algorithm. It focuses on the importance of agreeing on a single encoding and how complicated the state of affairs ends up being as a result of using different encodings today.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Architecture Board (IAB) and represents information that the IAB has deemed valuable to provide for permanent record. Documents approved for publication by the IAB are not a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6055.

Copyright Notice

Copyright (c) 2011 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1. Introduction 2
1.1. APIs 8
2. Use of Non-DNS Protocols 9
3. Use of Non-ASCII in DNS 10
3.1. Examples 14
4. Recommendations 16
5. Security Considerations 18
6. Acknowledgements 19
7. IAB Members at the Time of Approval 19
8. References 20
8.1. Normative References 20
8.2. Informative References 20

1. Introduction

The goal of this document is to explore what can be learned from some current difficulties in implementing Internationalized Domain Names (IDNs).

A domain name consists of a sequence of labels, conventionally written separated by dots. An IDN is a domain name that contains one or more labels that, in turn, contain one or more non-ASCII characters. Just as with plain ASCII domain names, each IDN label must be encoded using some mechanism before it can be transmitted in network packets, stored in memory, stored on disk, etc. These encodings need to be reversible, but they need not store domain names the same way humans conventionally write them on paper. For example, when transmitted over the network in DNS packets, domain name labels are *not* separated with dots.

Internationalized Domain Names for Applications (IDNA), discussed later in this document, is the standard that defines the use and coding of internationalized domain names for use on the public Internet [RFC5890]. An earlier version of IDNA [RFC3490] is now being phased out. Except where noted, the two versions are approximately the same with regard to the issues discussed in this document. However, some explanations appeared in the earlier documents that were no longer considered useful when the later revision was created; they are quoted here from the documents in which they appear. In addition, the terminology of the two versions differ somewhat; this document reflects the terminology of the current version.

Unicode [Unicode] is a list of characters (including non-spacing marks that are used to form some other characters), where each character is assigned an integer value, called a code point. In

simple terms a Unicode string is a string of integer code point values in the range 0 to 1,114,111 (10FFFF in base 16). These integer code points must be encoded using some mechanism before they can be transmitted in network packets, stored in memory, stored on disk, etc. Some common ways of encoding these integer code point values in computer systems include UTF-8, UTF-16, and UTF-32. In addition to the material below, those forms and the tradeoffs among them are discussed in Chapter 2 of The Unicode Standard [Unicode].

UTF-8 is a mechanism for encoding a Unicode code point in a variable number of 8-bit octets, where an ASCII code point is preserved as-is. Those octets encode a string of integer code point values, which represent a string of Unicode characters. The authoritative definition of UTF-8 is in Sections 3.9 and 3.10 of The Unicode Standard [Unicode], but a description of UTF-8 encoding can also be found in RFC 3629 [RFC3629]. Descriptions and formulae can also be found in Annex D of ISO/IEC 10646-1 [10646].

UTF-16 is a mechanism for encoding a Unicode code point in one or two 16-bit integers, described in detail in Sections 3.9 and 3.10 of The Unicode Standard [Unicode]. A UTF-16 string encodes a string of integer code point values that represent a string of Unicode characters.

UTF-32 (formerly UCS-4), also described in Sections 3.9 and 3.10 of The Unicode Standard [Unicode], is a mechanism for encoding a Unicode code point in a single 32-bit integer. A UTF-32 string is thus a string of 32-bit integer code point values, which represent a string of Unicode characters.

Note that UTF-16 results in some all-zero octets when code points occur early in the Unicode sequence, and UTF-32 always has all-zero octets.

IDNA specifies validity of a label, such as what characters it can contain, relationships among them, and so on, in Unicode terms. Valid labels can be in either "U-label" or "A-label" form, with the appropriate one determined by particular protocols or by context. U-label form is a direct representation of the Unicode characters using one of the encoding forms discussed above. This document discusses UTF-8 strings in many places. While all U-labels can be represented by UTF-8 strings, not all UTF-8 strings are valid U-labels (see Section 2.3.2 of the IDNA Definitions document [RFC5890] for a discussion of these distinctions). A-label form uses a compressed, ASCII-compatible encoding (an "ACE" in IDNA and other terminology) produced by an algorithm called Punycode. U-labels and

A-labels are duals of each other: transformations from one to the other do not lose information. The transformation mechanisms are specified in the IDNA Protocol document [RFC5891].

Punycode [RFC3492] is thus a mechanism for encoding a Unicode string in an ASCII-compatible encoding, i.e., using only letters, digits, and hyphens from the ASCII character set. When a Unicode label that is valid under the IDNA rules (a U-label) is encoded with Punycode for IDNA purposes, it is prefixed with "xn--"; the result is called an A-label. The prefix convention assumes that no other DNS labels (at least no other DNS labels in IDNA-aware applications) are allowed to start with these four characters. Consequently, when A-label encoding is assumed, any DNS labels beginning with "xn--" now have a different meaning (the Punycode encoding of a label containing one or more non-ASCII characters) or no defined meaning at all (in the case of labels that are not IDNA-compliant, i.e., are not well-formed A-labels).

ISO-2022-JP [RFC1468] is a mechanism for encoding a string of ASCII and Japanese characters, where an ASCII character is preserved as-is. ISO-2022-JP is stateful: special sequences are used to switch between character coding tables. As a result, if there are lost or mangled characters in a character stream, it is extremely difficult to recover the original stream after such a lost character encoding shift.

Comparison of Unicode strings is not as easy as comparing ASCII strings. First, there are a multitude of ways to represent a string of Unicode characters. Second, in many languages and scripts, the actual definition of "same" is very context-dependent. Because of this, comparison of two Unicode strings must take into account how the Unicode strings are encoded. Regardless of the encoding, however, comparison cannot simply be done by comparing the encoded Unicode strings byte by byte. The only time that is possible is when the strings are both mapped into some canonical form and encoded the same way.

In 1996 the IAB sponsored a workshop on character sets and encodings [RFC2130]. This document adds to that discussion and focuses on the importance of agreeing on a single encoding and how complicated the state of affairs ends up being as a result of using different encodings today.

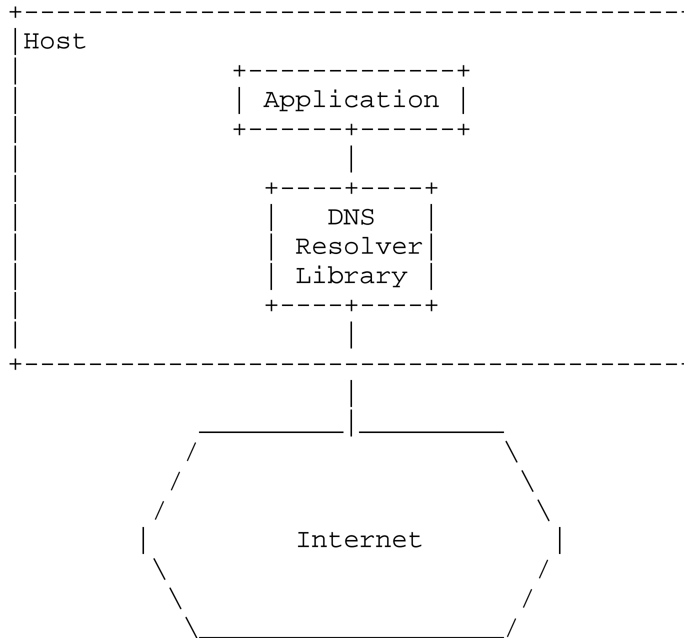
Different applications, APIs, and protocols use different encoding schemes today. Many of them were originally defined to use only ASCII. Internationalizing Domain Names in Applications (IDNA) [RFC5890] defines a mechanism that requires changes to applications, but in an attempt not to change APIs or servers, specifies that the

A-label format is to be used in many contexts. In some ways this could be seen as not changing the existing APIs, in the sense that the strings being passed to and from the APIs are still apparently ASCII strings. In other ways it is a very profound change to the existing APIs, because while those strings are still syntactically valid ASCII strings, they no longer mean the same thing that they used to. What looks like a plain ASCII string to one piece of software or library could be seen by another piece of software or library (with the application of out-of-band information) to be in fact an encoding of a Unicode string.

Section 1.3 of the original IDNA specification [RFC3490] states:

> The IDNA protocol is contained completely within applications. It is not a client-server or peer-to-peer protocol: everything is done inside the application itself. When used with a DNS resolver library, IDNA is inserted as a "shim" between the application and the resolver library. When used for writing names into a DNS zone, IDNA is used just before the name is committed to the zone.

Figure 1 depicts a simplistic architecture that a naive reader might assume from the paragraph quoted above. (A variant of this same picture appears in Section 6 of the original IDNA specification [RFC3490], further strengthening this assumption.)



Simplistic Architecture

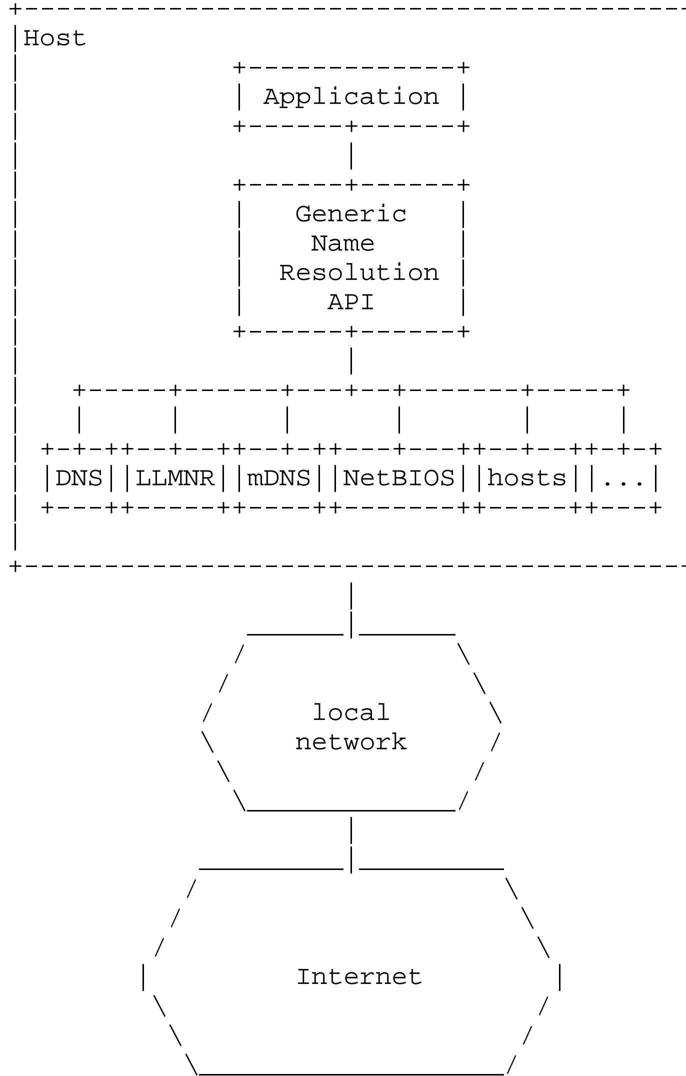
Figure 1

There are, however, two problems with this simplistic architecture that cause it to differ from reality.

First, resolver APIs on Operating Systems (OSs) today (Mac OS, Windows, Linux, etc.) are not DNS-specific. They typically provide a layer of indirection so that the application can work independent of the name resolution mechanism, which could be DNS, mDNS [DNS-MULTICAST], LLMNR [RFC4795], NetBIOS-over-TCP [RFC1001][RFC1002], hosts table [RFC0952], NIS [NIS], or anything else. For example, "Basic Socket Interface Extensions for IPv6" [RFC3493] specifies the getaddrinfo() API and contains many phrases like "For example, when using the DNS" and "any type of name resolution service (for example, the DNS)". Importantly, DNS is mentioned only as an example, and the application has no knowledge as to whether DNS or some other protocol will be used.

Second, even with the DNS protocol, private namespaces (sometimes including private uses of the DNS) do not necessarily use the same character set encoding scheme as the public Internet namespace.

We will discuss each of the above issues in subsequent sections. For reference, Figure 2 depicts a more realistic architecture on typical hosts today (which don't have IDNA inserted as a shim immediately above the DNS resolver library). More generally, the host may be attached to one or more local networks, each of which may or may not be connected to the public Internet and may or may not have a private namespace.



Realistic Architecture

Figure 2

1.1. APIs

Section 6.2 of the original IDNA specification [RFC3490] states (where ToASCII and ToUnicode below refer to conversions using the Punycode algorithm):

> It is expected that new versions of the resolver libraries in the future will be able to accept domain names in other charsets than ASCII, and application developers might one day pass not only domain names in Unicode, but also in local script to a new API for the resolver libraries in the operating system. Thus the ToASCII and ToUnicode operations might be performed inside these new versions of the resolver libraries.

Resolver APIs such as getaddrinfo() and its predecessor gethostbyname() were defined to accept C-Language "char *" arguments, meaning they accept a string of bytes, terminated with a NULL (0) byte. Because of the use of a NULL octet as a string terminator, this is sufficient for ASCII strings (including A-labels) and even ISO-2022-JP [RFC1468] and UTF-8 strings (unless an implementation artificially precludes them), but not UTF-16 or UTF-32 strings because a NULL octet could appear in the middle of strings using these encodings. Several operating systems historically used in Japan will accept (and expect) ISO-2022-JP strings in such APIs. Some platforms used worldwide also have new versions of the APIs (e.g., GetAddrInfoW() on Windows) that accept other encoding schemes such as UTF-16.

It is worth noting that an API using C-Language "char *" arguments can distinguish between conventional ASCII "hostname" labels, A-labels, ISO-2022-JP, and UTF-8 labels in names if the coding is known to be one of those four, and the label is intact (no lost or mangled characters). If a stateful encoding like ISO-2022-JP is used, applications extracting labels from text must take special precautions to be sure that the appropriate state-setting characters are included in the string passed to the API.

An example method for distinguishing among such codings is as follows:

- if the label contains an ESC (0x1B) byte, the label is ISO-2022-JP; otherwise,

- if any byte in the label has the high bit set, the label is UTF-8; otherwise,

- if the label starts with "xn--", then it is presumed to be an A-label; otherwise,

- o the label is ASCII (and therefore, by definition, the label is also UTF-8, since ASCII is a subset of UTF-8).

Again this assumes that ASCII labels never start with "xn--", and also that UTF-8 strings never contain an ESC character. Also the above is merely an illustration; UTF-8 can be detected and distinguished from other 8-bit encodings with good accuracy [MJD].

It is more difficult or impossible to distinguish the ISO 8859 character sets [ISO8859] from each other, because they differ in up to about 90 characters that have exactly the same encodings, and a short string is very unlikely to contain enough characters to allow a receiver to deduce the character set. Similarly, it is not possible in general to distinguish between ISO-2022-JP and any other encoding based on ISO 2022 code table switching.

Although it is possible (as in the example above) to distinguish some encodings when not explicitly specified, it is cleaner to have the encodings specified explicitly, such as specifying UTF-16 for GetAddrInfoW(), or specifying explicitly which APIs expect UTF-8 strings.

2. Use of Non-DNS Protocols

As noted earlier, typical name resolution libraries are not DNS-specific. Furthermore, some protocols are defined to use encoding forms other than IDNA A-labels. For example, mDNS [DNS-MULTICAST] specifies that UTF-8 be used. Indeed, the IETF policy on character sets and languages [RFC2277] (which followed the 1996 IAB-sponsored workshop [RFC2130]) states:

> Protocols MUST be able to use the UTF-8 charset, which consists of the ISO 10646 coded character set combined with the UTF-8 character encoding scheme, as defined in [10646] Annex R (published in Amendment 2), for all text.
>
> Protocols MAY specify, in addition, how to use other charsets or other character encoding schemes for ISO 10646, such as UTF-16, but lack of an ability to use UTF-8 is a violation of this policy; such a violation would need a variance procedure ([BCP9] section 9) with clear and solid justification in the protocol specification document before being entered into or advanced upon the standards track.
>
> For existing protocols or protocols that move data from existing datastores, support of other charsets, or even using a default other than UTF-8, may be a requirement. This is acceptable, but UTF-8 support MUST be possible.

Applications that convert an IDN to A-label form before calling getaddrinfo() will result in name resolution failures if the Punycode name is directly used in such protocols. Having libraries or protocols to convert from A-labels to the encoding scheme defined by the protocol (e.g., UTF-8) would require changes to APIs and/or servers, which IDNA was intended to avoid.

As a result, applications that assume that non-ASCII names are resolved using the public DNS and blindly convert them to A-labels without knowledge of what protocol will be selected by the name resolution library, have problems. Furthermore, name resolution libraries often try multiple protocols until one succeeds, because they are defined to use a common namespace. For example, the hosts file [RFC0952], NetBIOS-over-TCP [RFC1001], and DNS [RFC1034], are all defined to be able to share a common syntax. This means that when an application passes a name to be resolved, resolution may in fact be attempted using multiple protocols, each with a potentially different encoding scheme. For this to work successfully, the name must be converted to the appropriate encoding scheme only after the choice is made to use that protocol. In general, this cannot be done by the application since the choice of protocol is not made by the application.

3. Use of Non-ASCII in DNS

A common misconception is that DNS only supports names that can be expressed using letters, digits, and hyphens.

This misconception originally stems from the 1985 definition of an "Internet hostname" (and net, gateway, and domain name) for use in the "hosts" file [RFC0952]. An Internet hostname was defined therein as including only letters, digits, and hyphens, where uppercase and lowercase letters were to be treated as identical. The DNS specification [RFC1034], Section 3.5 entitled "Preferred name syntax" then repeated this definition in 1987, saying that this "syntax will result in fewer problems with many applications that use domain names (e.g., mail, TELNET)".

The confusion was thus left as to whether the "preferred" name syntax was a mandatory restriction in DNS, or merely "preferred".

The definition of an Internet hostname was updated in 1989 ([RFC1123], Section 2.1) to allow names starting with a digit. However, it did not address the increasing confusion as to whether all names in DNS are "hostnames", or whether a "hostname" is merely a special case of a DNS name.

By 1997, things had progressed to a state where it was necessary to clarify these areas of confusion. "Clarifications to the DNS Specification" [RFC2181], Section 11 states:

> The DNS itself places only one restriction on the particular labels that can be used to identify resource records. That one restriction relates to the length of the label and the full name. The length of any one label is limited to between 1 and 63 octets. A full domain name is limited to 255 octets (including the separators). The zero length full name is defined as representing the root of the DNS tree, and is typically written and displayed as ".". Those restrictions aside, any binary string whatever can be used as the label of any resource record. Similarly, any binary string can serve as the value of any record that includes a domain name as some or all of its value (SOA, NS, MX, PTR, CNAME, and any others that may be added). Implementations of the DNS protocols must not place any restrictions on the labels that can be used.

Hence, it clarified that the restriction to letters, digits, and hyphens does not apply to DNS names in general, nor to records that include "domain names". Hence, the "preferred" name syntax described in the original DNS specification [RFC1034] is indeed merely "preferred", not mandatory.

Since there is no restriction even to ASCII, let alone letter-digit-hyphen use, DNS does not violate the subsequent IETF requirement to allow UTF-8 [RFC2277].

Using UTF-16 or UTF-32 encoding, however, would not be ideal for use in DNS packets or C-Language "char *" APIs because existing software already uses ASCII, and UTF-16 and UTF-32 strings can contain all-zero octets that existing software will interpret as the end of the string. To use UTF-16 or UTF-32, one would need some way of knowing whether the string was encoded using ASCII, UTF-16, or UTF-32, and indeed for UTF-16 or UTF-32 whether it was big-endian or little-endian encoding. In contrast, UTF-8 works well because any 7-bit ASCII string is also a UTF-8 string representing the same characters.

If a private namespace is defined to use UTF-8 (and not other encodings such as UTF-16 or UTF-32), there's no need for a mechanism to know whether a string was encoded using ASCII or UTF-8, because (for any string that can be represented using ASCII) the representations are exactly the same. In other words, for any string that can be represented using ASCII, it doesn't matter whether it is interpreted as ASCII or UTF-8 because both encodings are the same, and for any string that can't be represented using ASCII, it's

obviously UTF-8. In addition, unlike UTF-16 and UTF-32, ASCII and UTF-8 are both byte-oriented encodings so the question of big-endian or little-endian encoding doesn't apply.

While implementations of the DNS protocol must not place any restrictions on the labels that can be used, applications that use the DNS are free to impose whatever restrictions they like, and many have. The above rules permit a domain name label that contains unusual characters, such as embedded spaces, which many applications consider a bad idea. For example, the original specification [RFC0821] of the SMTP protocol [RFC5321] constrains the character set usable in email addresses. There is now an effort underway to define an extension to SMTP to support internationalized email addresses and headers. See the EAI framework [RFC4952] for more discussion on this topic.

Shortly after the DNS Clarifications [RFC2181] and IETF character sets and languages policy [RFC2277] were published, the need for internationalized names within private namespaces (i.e., within enterprises) arose. The current (and past, predating IDNA and the prefixed ACE conventions) practice within enterprises that support other languages is to put UTF-8 names in their internal DNS servers in a private namespace. For example, "Using the UTF-8 Character Set in the Domain Name System" [UTF8-DNS] was first written in 1997, and was then widely deployed in Windows. The use of UTF-8 names in DNS was similarly implemented and deployed in Mac OS, simply by virtue of the fact that applications blindly passed UTF-8 strings to the name resolution APIs, the name resolution APIs blindly passed those UTF-8 strings to the DNS servers, and the DNS servers correctly answered those queries. From the user's point of view, everything worked properly without any special new code being written, except that ASCII is matched case-insensitively whereas UTF-8 is not (although some enterprise DNS servers reportedly attempt to do case-insensitive matching on UTF-8 within private namespaces, an action that causes other problems and violates a subsequent prohibition [RFC4343]). Within a private namespace, and especially in light of the IETF UTF-8 policy [RFC2277], it was reasonable to assume that binary strings were encoded in UTF-8.

As implied earlier, there are also issues with mapping strings to some canonical form, independent of the encoding. Such issues are not discussed in detail in this document. They are discussed to some extent in, for example, Section 3 of "Unicode Format for Network Interchange" [RFC5198], and are left as opportunities for elaboration in other documents.

A few years after UTF-8 was already in use in private namespaces in DNS, the strategy of using a reserved prefix and an ASCII-compatible

encoding (ACE) was developed for IDNA. That strategy included the Punycode algorithm, which began to be developed (during the period from 2002 [IDN-PUNYCODE] to 2003 [RFC3492]) for use in the public DNS namespace. There were a number of reasons for this. One such reason the prefixed ACE strategy was selected for the public DNS namespace had to do with the fact that other encodings such as ISO 8859-1 were also in use in DNS and the various encodings were not necessarily distinguishable from each other. Another reason had to do with concerns about whether the details of IDNA, including the use of the Punycode algorithm, were an adequate solution to the problems that were posed. If either the Punycode algorithm or fundamental aspects of character handling were wrong, and had to be changed to something incompatible, it would be possible to switch to a new prefix or adopt another model entirely. Only the part of the public DNS namespace that starts a label with "xn--" would be polluted.

Today the algorithm is seen as being about as good as it can realistically be, so moving to a different encoding (UTF-8 as suggested in this document) that can be viewed as "native" would not be as risky as it would have been in 2002.

In any case, the publication of Punycode [RFC3492] and the dependencies on it in the IDNA Protocol document [RFC5891] and the earlier IDNA specification [RFC3490] thus resulted in having to use different encodings for different namespaces (where UTF-8 for private namespaces was already deployed). Hence, referring back to Figure 2, a different encoding scheme may be in use on the Internet vs. a local network.

In general, a host may be connected to zero or more networks using private namespaces, plus potentially the public namespace. Applications that convert a U-label form IDN to an A-label before calling getaddrinfo() will incur name resolution failures if the name is actually registered in a private namespace in some other encoding (e.g., UTF-8). Having libraries or protocols convert from A-labels to the encoding used by a private namespace (e.g., UTF-8) would require changes to APIs and/or servers, which IDNA was intended to avoid.

Also, a fully-qualified domain name (FQDN) to be resolved may be obtained directly from an application, or it may be composed by the DNS resolver itself from a single label obtained from an application by using a configured suffix search list, and the resulting FQDN may use multiple encodings in different labels. For more information on the suffix search list, see Section 6 of "Common DNS Implementation Errors and Suggested Fixes" [RFC1536], the DHCP Domain Search Option [RFC3397], and Section 4 of "DNS Configuration options for DHCPv6" [RFC3646].

As noted in Section 6 of "Common DNS Implementation Errors and Suggested Fixes" [RFC1536], the community has had bad experiences (e.g., security problems [RFC1535]) with "searching" for domain names by trying multiple variations or appending different suffixes. Such searching can yield inconsistent results depending on the order in which alternatives are tried. Nonetheless, the practice is widespread and must be considered.

The practice of searching for names, whether by the use of a suffix search list or by searching in different namespaces, can yield inconsistent results. For example, even when a suffix search list is only used when an application provides a name containing no dots, two clients with different configured suffix search lists can get different answers, and the same client could get different answers at different times if it changes its configuration (e.g., when moving to another network). A deeper discussion of this topic is outside the scope of this document.

3.1. Examples

Some examples of cases that can happen in existing implementations today (where {non-ASCII} below represents some user-entered non-ASCII string) are:

- User types in {non-ASCII}.{non-ASCII}.com, and the application passes it, in the form of a UTF-8 string, to getaddrinfo() or gethostbyname() or equivalent.

   1. The DNS resolver passes the (UTF-8) string unmodified to a DNS server.

- User types in {non-ASCII}.{non-ASCII}.com, and the application passes it to a name resolution API that accepts strings in some other encoding such as UTF-16, e.g., GetAddrInfoW() on Windows.

   1. The name resolution API decides to pass the string to DNS (and possibly other protocols).

   2. The DNS resolver converts the name from UTF-16 to UTF-8 and passes the query to a DNS server.

- User types in {non-ASCII}.{non-ASCII}.com, but the application first converts it to A-label form such that the name that is passed to name resolution APIs is (say) xn--e1afmkfd.xn--80akhbyknj4f.com.

   1. The name resolution API decides to pass the string to DNS (and possibly other protocols).

2. The DNS resolver passes the string unmodified to a DNS server.

3. If the name is not found in DNS, the name resolution API decides to try another protocol, say mDNS.

4. The query goes out in mDNS, but since mDNS specified that names are to be registered in UTF-8, the name isn't found since it was encoded as an A-label in the query.

- User types in {non-ASCII}, and the application passes it, in the form of a UTF-8 string, to getaddrinfo() or equivalent.

  1. The name resolution API decides to pass the string to DNS (and possibly other protocols).

  2. The DNS resolver will append suffixes in the suffix search list, which may contain UTF-8 characters if the local network uses a private namespace.

  3. Each FQDN in turn will then be sent in a query to a DNS server, until one succeeds.

- User types in {non-ASCII}, but the application first converts it to an A-label, such that the name that is passed to getaddrinfo() or equivalent is (say) xn--e1afmkfd.

  1. The name resolution API decides to pass the string to DNS (and possibly other protocols).

  2. The DNS stub resolver will append suffixes in the suffix search list, which may contain UTF-8 characters if the local network uses a private namespace, resulting in (say) xn--e1afmkfd.{non-ASCII}.com

  3. Each FQDN in turn will then be sent in a query to a DNS server, until one succeeds.

  4. Since the private namespace in this case uses UTF-8, the above queries fail, since the A-label version of the name was not registered in that namespace.

- User types in {non-ASCII1}.{non-ASCII2}.{non-ASCII3}.com, where {non-ASCII3}.com is a public namespace using IDNA and A-labels, but {non-ASCII2}.{non-ASCII3}.com is a private namespace using UTF-8, which is accessible to the user. The application passes the name, in the form of a UTF-8 string, to getaddrinfo() or equivalent.

1. The name resolution API decides to pass the string to DNS (and possibly other protocols).

2. The DNS resolver tries to locate the authoritative server, but fails the lookup because it cannot find a server for the UTF-8 encoding of {non-ASCII3}.com, even though it would have access to the private namespace. (To make this work, the private namespace would need to include the UTF-8 encoding of {non-ASCII3}.com.)

When users use multiple applications, some of which do A-label conversion prior to passing a name to name resolution APIs, and some of which do not, odd behavior can result which at best violates the Principle of Least Surprise, and at worst can result in security vulnerabilities.

First consider two competing applications, such as web browsers, that are designed to achieve the same task. If the user types the same name into each browser, one may successfully resolve the name (and hence access the desired content) because the encoding scheme is correct, while the other may fail name resolution because the encoding scheme is incorrect. Hence the issue can incent users to switch to another application (which in some cases means switching to an IDNA application, and in other cases means switching away from an IDNA application).

Next consider two separate applications where one is designed to be launched from the other, for example a web browser launching a media player application when the link to a media file is clicked. If both types of content (web pages and media files in this example) are hosted at the same IDN in a private namespace, but one application converts to A-labels before calling name resolution APIs and the other does not, the user may be able to access a web page, click on the media file causing the media player to launch and attempt to retrieve the media file, which will then fail because the IDN encoding scheme was incorrect. Or even worse, if an attacker is able to register the same name in the other encoding scheme, the user may get the content from the attacker's machine. This is similar to a normal phishing attack, except that the two names represent exactly the same Unicode characters.

4. Recommendations

On many platforms, the name resolution library will automatically use a variety of protocols to search a variety of namespaces, which might be using UTF-8 or other encodings. In addition, even when only the DNS protocol is used, in many operational environments, a private DNS

namespace using UTF-8 is also deployed and is automatically searched by the name resolution library.

As explained earlier, using multiple canonical formats, and multiple encodings in different protocols or even in different places in the same namespace creates problems. Because of this, and the fact that both IDNA A-labels and UTF-8 are in use as encoding mechanisms for domain names today, we make the recommendations described below.

It is inappropriate for an application that calls a general-purpose name resolution library to convert a name to an A-label unless the application is absolutely certain that, in all environments where the application might be used, only the global DNS that uses IDNA A-labels actually will be used to resolve the name.

Instead, conversion to A-label form, or any other special encoding required by a particular name-lookup protocol, should be done only by an entity that knows which protocol will be used (e.g., the DNS resolver, or getaddrinfo() upon deciding to pass the name to DNS), rather than by general applications that call protocol-independent name resolution APIs. (Of course, applications that store strings internally in a different format than that required by those APIs, need to convert strings from their own internal format to the format required by the API.) Similarly, even if an application can know that DNS is to be used, the conversion to A-labels should be done only by an entity that knows which part of the DNS namespace will be used.

That is, a more intelligent DNS resolver would be more liberal in what it would accept from an application and be able to query for both a name in A-label form (e.g., over the Internet) and a UTF-8 name (e.g., over a corporate network with a private namespace) in case the server only recognizes one. However, we might also take into account that the various resolution behaviors discussed earlier could also occur with record updates (e.g., with Dynamic Update [RFC2136]), resulting in some names being registered in a local network's private namespace by applications doing conversion to A-labels, and other names being registered using UTF-8. Hence, a name might have to be queried with both encodings to be sure to succeed without changes to DNS servers.

Similarly, a more intelligent stub resolver would also be more liberal in what it would accept from a response as the value of a record (e.g., PTR) in that it would accept either UTF-8 (U-labels in the case of IDNA) or A-labels and convert them to whatever encoding is used by the application APIs to return strings to applications.

Indeed the choice of conversion within the resolver libraries is consistent with the quote from Section 6.2 of the original IDNA specification [RFC3490] stating that conversion using the Punycode algorithm (i.e., to A-labels) "might be performed inside these new versions of the resolver libraries".

That said, some application-layer protocols (e.g., EPP Domain Name Mapping [RFC5731]) are defined to use A-labels rather than simply using UTF-8 as recommended by the IETF character sets and languages policy [RFC2277]. In this case, an application may receive a string containing A-labels and want to pass it to name resolution APIs. Again the recommendation that a resolver library be more liberal in what it would accept from an application would mean that such a name would be accepted and re-encoded as needed, rather than requiring the application to do so.

It is important that any APIs used by applications to pass names specify what encoding(s) the API uses. For example, GetAddrInfoW() on Windows specifies that it accepts UTF-16 and only UTF-16. In contrast, the original specification of getaddrinfo() [RFC3493] does not, and hence platforms vary in what they use (e.g., Mac OS uses UTF-8 whereas Windows uses Windows code pages).

Finally, the question remains about what, if anything, a DNS server should do to handle cases where some existing applications or hosts do IDNA queries using A-labels within the local network using a private namespace, and other existing applications or hosts send UTF-8 queries. It is undesirable to store different records for different encodings of the same name, since this introduces the possibility for inconsistency between them. Instead, a new DNS server serving a private namespace using UTF-8 could potentially treat encoding-conversion in the same way as case-insensitive comparison which a DNS server is already required to do, as long the DNS server has some way to know what the encoding is. Two encodings are, in this sense, two representations of the same name, just as two case-different strings are. However, whereas case comparison of non-ASCII characters is complicated by ambiguities (as explained in the IAB’s Review and Recommendations for Internationalized Domain Names [RFC4690]), encoding conversion between A-labels and U-labels is unambiguous.

5. Security Considerations

Having applications convert names to prefixed ACE format (A-labels) before calling name resolution can result in security vulnerabilities. If the name is resolved by protocols or in zones for which records are registered using other encoding schemes, an attacker can claim the A-label version of the same name and hence

trick the victim into accessing a different destination. This can be done for any non-ASCII name, even when there is no possible confusion due to case, language, or other issues. Other types of confusion beyond those resulting simply from the choice of encoding scheme are discussed in "Review and Recommendations for IDNs" [RFC4690].

Designers and users of encodings that represent Unicode strings in terms of ASCII should also consider whether trademark protection or phishing are issues, e.g., if one name would be encoded in a way that would be naturally associated with another organization or product.

6. Acknowledgements

The authors wish to thank Patrik Faltstrom, Martin Duerst, JFC Morfin, Ran Atkinson, S. Moonesamy, Paul Hoffman, and Stephane Bortzmeyer for their careful review and helpful suggestions. It is also interesting to note that none of the first three individuals' names above can be spelled out and written correctly in ASCII text. Furthermore, one of the IAB member's names below (Andrei Robachevsky) cannot be written in the script as it appears on his birth certificate.

7. IAB Members at the Time of Approval

Bernard Aboba
Marcelo Bagnulo
Ross Callon
Spencer Dawkins
Vijay Gill
Russ Housley
John Klensin
Olaf Kolkman
Danny McPherson
Jon Peterson
Andrei Robachevsky
Dave Thaler
Hannes Tschofenig

8. References

8.1. Normative References

[10646] International Organization for Standardization, "Information Technology - Universal Multiple-octet coded Character Set (UCS)".

ISO/IEC Standard 10646, comprised of ISO/IEC 10646-1:2000, "Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane", ISO/IEC 10646-2:2001, "Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 2: Supplementary Planes" and ISO/IEC 10646- 1:2000/Amd 1:2002, "Mathematical symbols and other characters".

[Unicode] The Unicode Consortium. The Unicode Standard, Version 5.1.0, defined by: "The Unicode Standard, Version 5.0", Boston, MA, Addison-Wesley, 2007, ISBN 0-321-48091-0, as amended by Unicode 5.1.0 (http://www.unicode.org/versions/Unicode5.1.0/).

8.2. Informative References

[DNS-MULTICAST] Cheshire, S. and M. Krochmal, "Multicast DNS", Work in Progress, February 2011.

[IDN-PUNYCODE] Costello, A., "Punycode version 0.3.3", Work in Progress, January 2002.

[ISO8859] International Organization for Standardization, "Information technology -- 8-bit single-byte coded graphic character sets".

ISO/IEC Standard 8859, comprised of ISO/IEC 8859-1:1998, Part 1: Latin alphabet No. 1 - ISO/IEC 8859-2:1999, Part 2: Latin alphabet No. 2 - ISO/IEC 8859-3:1999, Part 3: Latin alphabet No. 3 - ISO/IEC 8859-4:1998, Part 4: Latin alphabet No. 4 - ISO/IEC 8859-5:1999, Part 5: Latin/Cyrillic alphabet - ISO/IEC 8859-6:1999, Part 6: Latin/Arabic alphabet - ISO/IEC 8859-7:2003, Part 7: Latin/Greek alphabet - ISO/IEC 8859-8:1999, Part 8: Latin/Hebrew alphabet - ISO/IEC 8859-9:1999, Part 9: Latin alphabet No. 5 - ISO/IEC 8859-10:1998, Part 10: Latin alphabet No. 6 - ISO/IEC 8859-11:2001, Part 11: Latin/Thai alphabet - ISO/IEC 8859-13:1998, Part 13: Latin alphabet No. 7

- ISO/IEC 8859-14:1998, Part 14: Latin alphabet No. 8 (Celtic) - ISO/IEC 8859-15:1999, Part 15: Latin alphabet No. 9 - ISO/IEC 8859-16:2001, Part 16: Latin alphabet No. 10.

[MJD] Duerst, M., "The Properties and Promizes of UTF-8", 11th International Unicode Conference, San Jose , September 1997, <http://www.ifi.unizh.ch/mml/mduerst/papers/PDF/IUC11-UTF-8.pdf>.

[NIS] Sun Microsystems, "System and Network Administration", March 1990.

[RFC0821] Postel, J., "Simple Mail Transfer Protocol", STD 10, RFC 821, August 1982.

[RFC0952] Harrenstien, K., Stahl, M., and E. Feinler, "DoD Internet host table specification", RFC 952, October 1985.

[RFC1001] NetBIOS Working Group, "Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods", STD 19, RFC 1001, March 1987.

[RFC1002] NetBIOS Working Group, "Protocol standard for a NetBIOS service on a TCP/UDP transport: Detailed specifications", STD 19, RFC 1002, March 1987.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, November 1987.

[RFC1123] Braden, R., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, October 1989.

[RFC1468] Murai, J., Crispin, M., and E. van der Poel, "Japanese Character Encoding for Internet Messages", RFC 1468, June 1993.

[RFC1535] Gavron, E., "A Security Problem and Proposed Correction With Widely Deployed DNS Software", RFC 1535, October 1993.

[RFC1536] Kumar, A., Postel, J., Neuman, C., Danzig, P., and S. Miller, "Common DNS Implementation Errors and Suggested Fixes", RFC 1536, October 1993.

[RFC2130] Weider, C., Preston, C., Simonsen, K., Alvestrand, H., Atkinson, R., Crispin, M., and P. Svanberg, "The Report of the IAB Character Set Workshop held 29 February - 1 March, 1996", RFC 2130, April 1997.

[RFC2136] Vixie, P., Thomson, S., Rekhter, Y., and J. Bound, "Dynamic Updates in the Domain Name System (DNS UPDATE)", RFC 2136, April 1997.

[RFC2181] Elz, R. and R. Bush, "Clarifications to the DNS Specification", RFC 2181, July 1997.

[RFC2277] Alvestrand, H., "IETF Policy on Character Sets and Languages", BCP 18, RFC 2277, January 1998.

[RFC3397] Aboba, B. and S. Cheshire, "Dynamic Host Configuration Protocol (DHCP) Domain Search Option", RFC 3397, November 2002.

[RFC3490] Faltstrom, P., Hoffman, P., and A. Costello, "Internationalizing Domain Names in Applications (IDNA)", RFC 3490, March 2003.

[RFC3492] Costello, A., "Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)", RFC 3492, March 2003.

[RFC3493] Gilligan, R., Thomson, S., Bound, J., McCann, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 3493, February 2003.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC3646] Droms, R., "DNS Configuration options for Dynamic Host Configuration Protocol for IPv6 (DHCPv6)", RFC 3646, December 2003.

[RFC4343] Eastlake, D., "Domain Name System (DNS) Case Insensitivity Clarification", RFC 4343, January 2006.

[RFC4690] Klensin, J., Faltstrom, P., Karp, C., and IAB, "Review and Recommendations for Internationalized Domain Names (IDNs)", RFC 4690, September 2006.

[RFC4795] Aboba, B., Thaler, D., and L. Esibov, "Link-local Multicast Name Resolution (LLMNR)", RFC 4795, January 2007.

[RFC4952] Klensin, J. and Y. Ko, "Overview and Framework for Internationalized Email", RFC 4952, July 2007.

[RFC5198] Klensin, J. and M. Padlipsky, "Unicode Format for Network Interchange", RFC 5198, March 2008.

[RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, October 2008.

[RFC5731] Hollenbeck, S., "Extensible Provisioning Protocol (EPP) Domain Name Mapping", STD 69, RFC 5731, August 2009.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, August 2010.

[RFC5891] Klensin, J., "Internationalized Domain Names in Applications (IDNA): Protocol", RFC 5891, August 2010.

[UTF8-DNS] Kwan, S. and J. Gilroy, "Using the UTF-8 Character Set in the Domain Name System", Work in Progress, November 1997.

Authors' Addresses

Dave Thaler
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
USA

Phone: +1 425 703 8835
EMail: dthaler@microsoft.com

John C Klensin
1770 Massachusetts Ave, Ste 322
Cambridge, MA 02140

Phone: +1 617 245 1457
EMail: john+ietf@jck.com

Stuart Cheshire
Apple Inc.
1 Infinite Loop
Cupertino, CA 95014

Phone: +1 408 974 3207
EMail: cheshire@apple.com