Network Working Group
Internet-Draft
Intended status: Informational
Expires: May 14, 2010

D. Thaler
Microsoft
J. Klensin

S. Cheshire
Apple

November 10, 2009

IAB Thoughts on Encodings for Internationalized Domain Names
draft-iab-idn-encoding-01.txt

Abstract

This document explores issues with Internationalized Domain Names (IDNs) that result from the use of various encoding schemes such as Punycode and UTF-8.

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on May 14, 2010.

Table of Contents

1. Introduction

The goal of this document is to explore what can be learned from some current difficulties in implementing Internationalized Domain Names (IDNs). Although some elements of this exploration may immediately feed back into current IETF work, it is explicitly not the intention for this document to influence any current working group charter.

A domain name consists of a set of labels, conventionally written separated with dots. An Internationalized Domain Name (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.

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. It is defined in several documents, with the primary one of those being "Internationalizing Domain Names in Applications (IDNA)" [RFC3490]. A revision to the IDNA Standard is undergoing IETF Last Call review as this document is being written. That revision is reflected in [IDNA2008-Defs] and associated materials. Except where noted, the two versions are approximately the same with regard to the issues discussed in this document. However, their terminology differs somewhat; this document reflects the terminology of the earlier version.

Punycode [RFC3492] is a mechanism for encoding a Unicode [Unicode] string in ASCII characters using only letters, digits, and hyphens. When a Unicode label is encoded with Punycode, it is prefixed with "xn--", which assumes that other DNS labels are no longer allowed to start with these four characters. Consequently, when Punycode 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 well-formed Punycode).

The term "ToASCII" refers to the process of encoding a label containing one or more non-ASCII characters as an ASCII string beginning with "xn--". It consists of a combination of a non-reversible character mapping operation (e.g., converting upper case characters to lower case characters), plus a reversible encoding algorithm ('Punycode') that encodes a sequence of Unicode code points (which may contain code points above 127) as a sequence of ASCII code

points (containing only ASCII code points for letters, digits and hyphens). The term "ToUnicode" refers to the process of reversing the Punycode encoding, but not reversing the (irreversible) character mapping operation.

ISO-2022-JP [RFC1468] is a mechanism for encoding a string of ASCII and Japanese characters, where an ASCII character is preserved as-is.

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), which represent a string of Unicode characters. 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 [RFC3629] 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.

UTF-16 (formerly UCS-2) 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 [Unicode] Sections 3.9 and 3.10, 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 and UTF-32 codings result in some all-zero octets when code points occur early in the Unicode sequence.

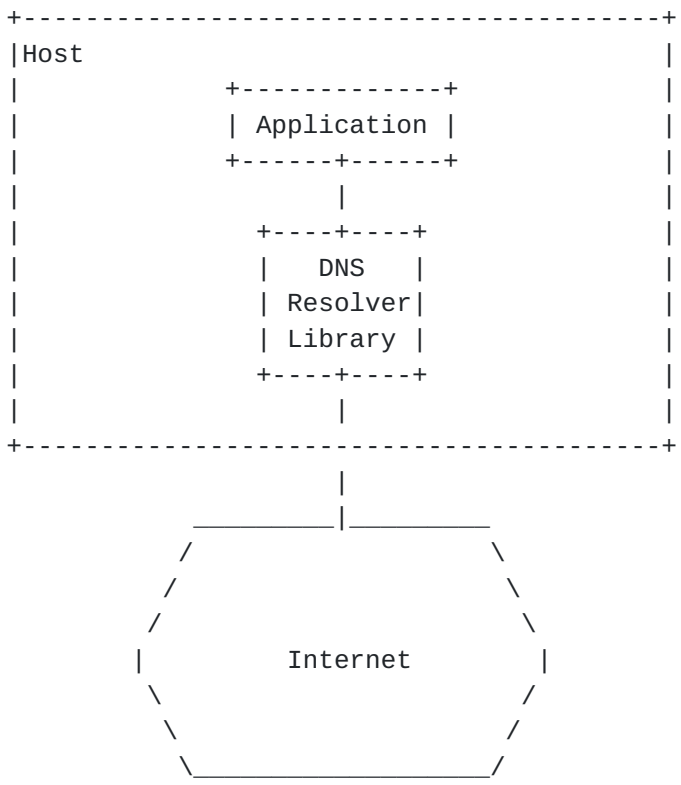
Different applications, APIs, and protocols use different encoding schemes today. Historically, many of them were originally defined to use only ASCII. Internationalizing Domain Names in Applications (IDNA) [RFC3490] defined a mechanism that required changes to applications, but in attempt not to change APIs or servers, specified that Punycode is to be used. 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 were still apparently ASCII strings. In

other ways it was a very profound change to the existing APIs, because while those strings were still syntactically valid ASCII strings, they no longer meant the same thing as they used to. What looked 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 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 IDNA specification [RFC3490] further strengthening this assumption.)



Simplistic Architecture

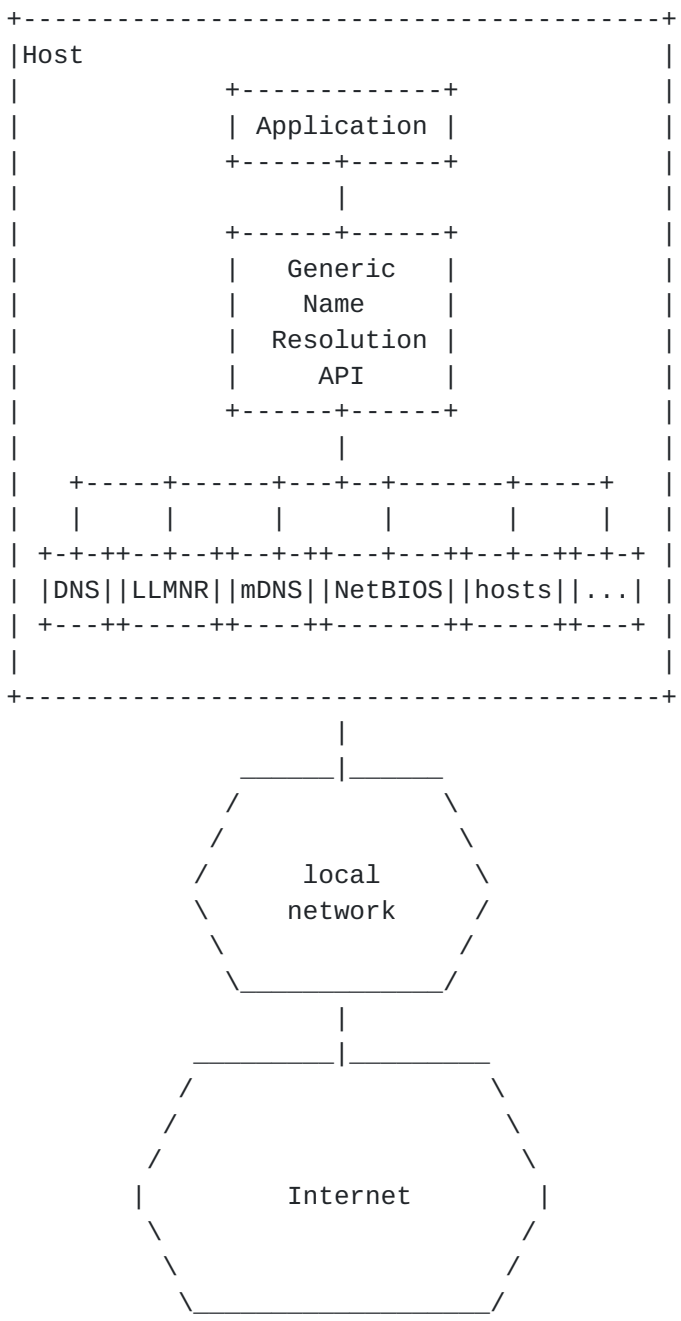
Figure 1

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

First, resolver APIs on Operating Systems (OSs) today (MacOS, 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 [I-D.cheshire-dnsext-multicastdns], LLMNR [RFC4795], NetBIOS-over-TCP [RFC1001][RFC1002], etc/hosts file [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 name spaces (sometimes including private uses of the DNS), do not necessarily use the same character set encoding scheme as the public Internet name space.

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 name space.



Realistic Architecture

Figure 2

1.1. APIs

Section 6.2 of the IDNA specification [RFC3490] states:

> 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 "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, Punycode strings, and even ISO-2022-JP and UTF-8 strings (unless an implementation artificially precludes them), but not UTF-16 or UTF-32 strings. 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 "char *" arguments can distinguish between ASCII, Punycode, ISO-2022-JP, and UTF-8 labels in names if the coding is known to be one of those four. An example method 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 contains a string in Punycode encoding; otherwise,
- the label is ASCII.

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 high precision [MJD].

It is more difficult or impossible to distinguish the ISO 8859 character sets from each other. 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 schemes other than Punycode. For example, mDNS [I-D.cheshire-dnsext-multicastdns] specifies that UTF-8 be used. Indeed, the IETF policy on character sets and languages [RFC2277] 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 Punycode 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 Punycode 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 Punycode 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 name space. For example, the hosts file, DNS, and NetBIOS-over-TCP are all defined to be able to share a common syntax (e.g., see ([RFC0952], [RFC1001] section 11.1.1, and [RFC1034] section 2.1). 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 stemmed from the definition in 1985 of an "Internet host name" (and net, gateway, and domain name) for use in the "hosts" file [RFC0952]. An Internet host name was defined therein as including only letters, digits, and hyphens, where upper and lower case 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 host name was updated in 1989 ([RFC1123] section 2.1) to allow names starting with a digit (to support IPv4 addresses in dotted-decimal form). Section 6.1 of "Requirements for Internet Hosts -- Application and Support" [RFC1123] discusses the use of DNS (and the hosts file) for resolving host names to IP addresses and vice versa. This led to confusion as to whether all names in DNS are "host names", or whether a "host name" 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 is in conformance with the 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 APIs because existing software already uses ASCII, and UTF-16 and UTF-32 strings can contain all-zero octets that existing software may 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 name space 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 would consider a bad idea. For example, the SMTP protocol [RFC5321], but going back to the original specification in [RFC0821], constrains the character set usable in email addresses. There is now an effort underway to permit SMTP to support internationalized email addresses via an extension.

Shortly after the DNS Clarifications [RFC2181] and IETF character sets and languages policy [RFC2277] were published, the need for internationalized names within private name spaces (i.e., within enterprises) arose. The current (and past, predating Punycode) practice within enterprises that support other languages is to put UTF-8 names in their internal DNS servers in a private name space. For example, "Using the UTF-8 Character Set in the Domain Name

System" [I-D.skwan-utf8-dns-00] 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 MacOS, simply by virtue of the fact that applications blindly passed UTF-8 strings to the name resolution APIs, and the name resolution APIs blindly passed those UTF-8 strings to the DNS servers, and the DNS servers correctly answered those queries, and 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 name spaces). Within a private name space, and especially in light of the IETF UTF-8 policy [RFC2277], it was reasonable to assume within a private name space that binary strings were encoded in UTF-8.

[EDITOR'S NOTE: There are also normalization/mapping issues. Currently we only explore encoding issues.]

Five years after UTF-8 was already in use in private name spaces in DNS, Punycode began to be developed (during the period from 2002 [I-D.ietf-idn-punycode-00] to 2003 [RFC3492]) for use in the public DNS name space. This publication thus resulted in having to use different encodings for different name spaces (where UTF-8 for private name spaces 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 name spaces, plus potentially the public name space. Applications that convert an IDN to Punycode before calling getaddrinfo() will result in name resolution failures if the name is actually registered in a private name space in some other encoding (e.g., UTF-8). Having libraries or protocols convert from Punycode to the encoding used by a private name space (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 [RFC1536] section 6, the community has had bad

experiences 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:

1. 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.
   * The DNS resolver passes the (UTF-8) string unmodified to a DNS server.
2. 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.
   * The name resolution API decides to pass the string to DNS (and possibly other protocols).
   * The DNS resolver converts the name from UTF-16 to UTF-8 and passes the query to a DNS server.
3. User types in {non-ASCII}.{non-ASCII}.com, but the application first converts it to Punycode such that the name that is passed to name resolution APIs is (say) xn--e1afmkfd.xn--80akhbyknj4f.com.
   * The name resolution API decides to pass the string to DNS (and possibly other protocols).
   * The DNS resolver passes the string unmodified to a DNS server.
   * If the name is not found in DNS, the name resolution API decides to try another protocol, say mDNS.
   * 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 Punycode encoded in the query.
4. User types in {non-ASCII}, and the application passes it, in the form of a UTF-8 string, to getaddrinfo or equivalent.

   * The name resolution API decides to pass the string to DNS (and possibly other protocols).
   * The DNS resolver will append suffixes in the suffix search list, which may contain UTF-8 characters if the local network uses a private name space.
   * Each FQDN in turn will then be sent in a query to a DNS server, until one succeeds.

5. User types in {non-ASCII}, but the application first converts it to Punycode, such that the name that is passed to getaddrinfo or equivalent is (say) xn--e1afmkfd.
   * The name resolution API decides to pass the string to DNS (and possibly other protocols).
   * 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 name space, resulting in (say) xn--e1afmkfd.{non-ASCII}.com
   * Each FQDN in turn will then be sent in a query to a DNS server, until one succeeds.
   * Since the private name space in this case uses UTF-8, the above queries fail, since the Punycode version of the name was not registered in that name space.
6. User types in {non-ASCII1}.{non-ASCII2}.{non-ASCII3}.com, where {non-ASCII3}.com is a public name space using Punycode, but {non-ASCII2}.{non-ASCII3}.com is a private name space 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.
   * The name resolution API decides to pass the string to DNS (and possibly other protocols).
   * 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 name space. (To make this work, the private name space would need to include the UTF-8 encoding of {non-ASCII3}.com.)

When users use multiple applications, some of which do Punycode 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 was correct, while the other may fail name resolution because the encoding scheme was 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 name space, but one application converts to Punycode 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 was able to register the same name in the other encoding scheme, 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

Taking into account the issues above, it would seem inappropriate for an application to convert a name to Punycode when it does not know whether DNS will be used by the name resolution library, or whether the name exists in a private name space that uses UTF-8, or in the global DNS that uses Punycode.

Instead, conversion to Punycode, UTF-8, or whatever other encoding, 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, it is still necessary for applications to convert to whatever form those APIs expect.) Similarly, even when DNS is used, the conversion to Punycode should be done only by an entity that knows which name space 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 Punycode name (e.g., over the Internet) and a UTF-8 name (e.g., over a corporate network with a private name space) in case the server only recognized 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 name space by applications doing Punycode conversion, 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 or Punycode 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 IDNA specification [RFC3490] stating that Punycode conversion "might be performed inside these new versions of the resolver libraries".

That said, some application-layer protocols may be defined to use Punycode rather than UTF-8 as recommended by the IETF character sets and languages policy [RFC2277]. In this case, an application may receive a Punycode name 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.

Finally, the question remains about what, if anything, a DNS server should do to handle cases where some existing applications or hosts do Punycode queries within the local network using a private name space, 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 name space 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 Punycode and UTF-8 is unambiguous.

[EDITOR'S NOTE: There are also normalization/mapping issues. Currently we only explore encoding issues.]

5. Security Considerations

Having applications convert names to Punycode 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 Punycode 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 is an issue, e.g., if one name would be encoded in a way that would be naturally associated with another organization, such as xn--rfc-editor.

6. IANA Considerations

[RFC Editor: please remove this section prior to publication.]

This document has no IANA Actions.

7. IAB Members at the time of this writing

Marcelo Bagnulo
Gonzalo Camarillo
Stuart Cheshire
Vijay Gill
Russ Housley
John Klensin
Olaf Kolkman
Gregory Lebovitz
Andrew Malis
Danny McPherson
David Oran
Jon Peterson
Dave Thaler

8. References

8.1. Normative References

[Unicode] The Unicode Consortium, "The Unicode Standard, Version 5.1.0", 2008.

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

[I-D.cheshire-dnsext-multicastdns]
Cheshire, S. and M. Krochmal, "Multicast DNS", draft-cheshire-dnsext-multicastdns-08 (work in progress), September 2009.

[I-D.ietf-idn-punycode-00]
Costello, A., "Punycode version 0.3.3", draft-ietf-idn-punycode-00 (work in progress), July 2002.

[I-D.skwan-utf8-dns-00]
Kwan, S. and J. Gilroy, "Using the UTF-8 Character Set in the Domain Name System", draft-skwan-utf8-dns-00 (work in progress), November 1997.

[IDNA2008-Defs]
Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", August 2009, <https://datatracker.ietf.org/drafts/draft-ietf-idnabis-defs/>.

[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.

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

[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.

[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.

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

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