Host/Split: Exploitable Antipatterns in Unicode Normalization

Jonathan Birch (jobirch@microsoft.com)

Abstract

This document describes "HostSplit" and "HostBond", new exploit techniques that leverage Unicode normalization to bypass URL security filters and, in some cases, allow one domain to impersonate another. Where previous attacks against internationalized domain names relied on visual spoofing, these attacks fool software with URL strings that are parsed as belonging to one hostname but resolved as belonging to a different host name.

The vulnerabilities that enable these attacks are widespread, because they result from practical compromises in implementing IDNA standards. The author of this paper identified several new vulnerabilities, including vulnerabilities in Edge/IE, .NET, Python, Java, Office 365, and Gmail. The Office 365 and Gmail vulnerabilities are discussed in this paper as examples. A more general exploit pattern against OAuth is also described.

Although some platform-level problems have already been corrected, many of the fixes for these vulnerabilities will need to be made in applications. It is likely that there are still many software packages with Unicode normalization vulnerabilities of this type. This paper discusses methods to test for these vulnerabilities as well as coding and design best practices for preventing them.

Contents

Background

The use of Unicode in hostnames was a late addition to the Internet. The "Internationalizing Domain Names in Applications" (IDNA) standard, which defines how Unicode hostnames should be processed, was only issued in 2003. By this time the Internet and the World Wide Web were already well-established. Because of the difficulties that would be involved in integrating Unicode into every Internet protocol, IDNA was designed as a conversion mechanism in the application layer, allowing lower-level protocols to continue to be ASCII-only.

The IDNA standard defines each hostname as having two forms: a Unicode label, or "U-label", which can contain Unicode characters and is used for display to users, and an ASCII label, or "A-label", which is used for DNS and other ASCII-only protocols. The standard also defines processes for converting between these labels. The exploits described in this document primarily relate to unexpected consequences of these conversion processes.

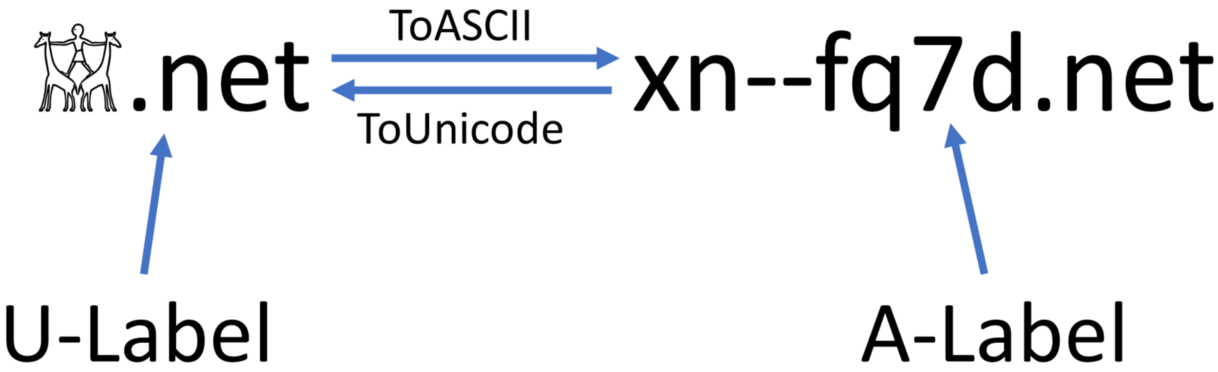

Figure 1 The relationship between U-label and A-label

Conversion from U-label to A-label, or “ToASCII”, is a two-step process:

1. Normalization
   Any Unicode characters in the hostname are converted to their "Compatibility Composition" (KC) normalization form. This is necessary because different combinations of Unicode characters can have identical visual representations. This normalization step is meant to ensure that hostnames that are rendered identically are also encoded identically.

2. Punycoding
   After normalization, an encoding algorithm called "Punycode" is applied to hostnames. This results in an ASCII string with three components: An "ASCII Compatible Encoding Prefix" (ACE) which consists of the string "xn--", followed by any ASCII characters in the normalized hostname in their original order, followed by a sequence of ASCII characters that act as instructions for a state machine that can be executed to reconstruct the non-ASCII portions of the hostname.

This process is repeated independently on each segment of the hostname. As an example, the U-label"fiskmås.net" becomes the A-label "xn--fiskms-mua.net". The "xn--" at the beginning is the ACE, followed by the ASCII characters from the hostname, followed by the state machine instructions "-mua" for reinserting the "å" between the "m" and the "s".

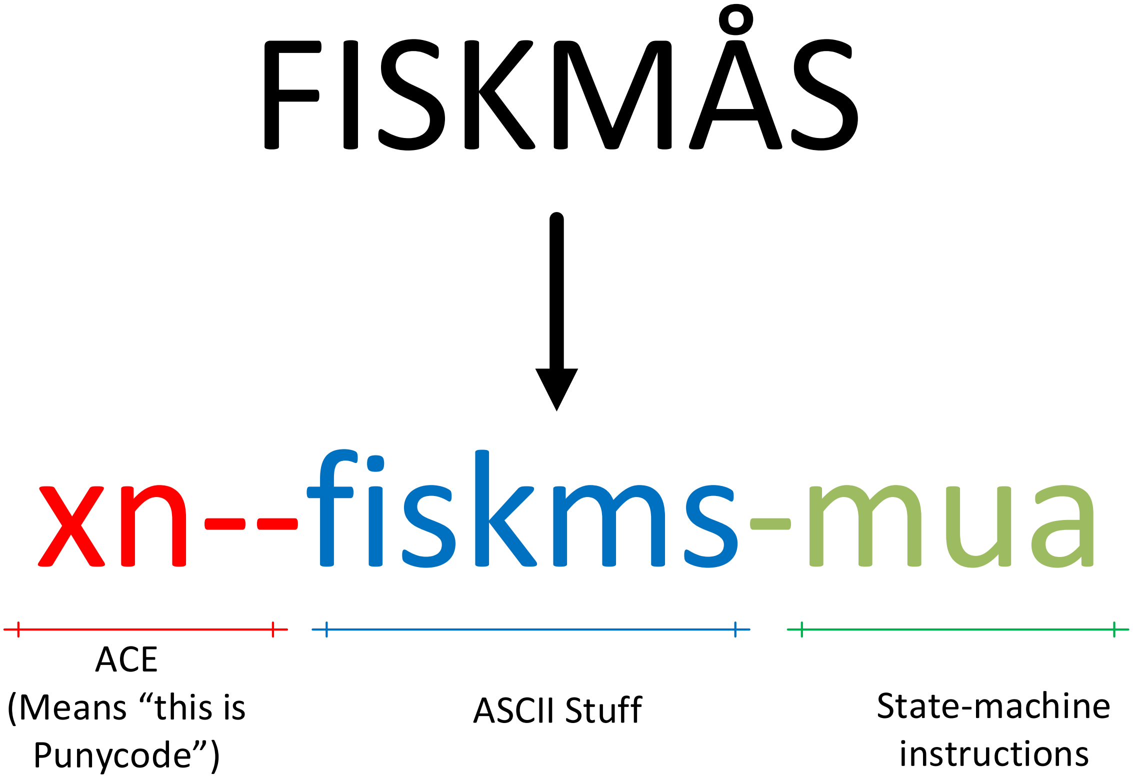

Figure 2 Punycode Anatomy

Conversion from A-label to U-label, or “ToUnicode”, is much simpler: the Punycode state machine is executed to reinsert any Unicode characters into the hostname, and the ACE is removed. No attempt is made to reverse the normalization step.

There are three distinct specifications that define how IDNA conversions should work: IDNA2003, IDNA2008, and IDNA2008 + UTS46. Each of these allows a different set of Unicode characters in U-labels and each defines a different set of normalization rules.

The HostSplit Vulnerability

Some Unicode characters have a KC normalization form consisting only of one or more ASCII characters. When a Unicode hostname consisting only of ASCII characters and Unicode characters that normalize to ASCII is converted to an A-label, no Punycoding is performed, as the normalization step removes all Unicode.

A small set of Unicode characters have a KC normalization form containing ASCII characters with syntax-significance in a protocol. As an example, the character "℀" (U+2100, "Account Of") normalizes to the

ASCII string "a/c". This is problematic because the forward slash character is used as a separator between a host name and a path in HTTP URL's. Some IDNA implementations will convert a U-label such as "www.evil.c℀.microsoft.com" to the A-label "www.evil.ca/c.microsoft.com". If such an A-label is integrated into a URL, the addition of the forward slash character has the effect of pushing part of the host name into the path, effectively changing the host name.

Many applications and application frameworks that parse URL's accept Unicode hostnames but do not convert them to A-labels before performing security checks. This means that they may parse a URL like "https://www.evil.c℀.microsoft.com" as a subdomain of "microsoft.com". However, the hostname of this URL must be converted to an A-label before a DNS lookup can be performed on it as part of making a request. In implementations where this URL becomes "https://www.evil.ca/c.microsoft.com" as a result of Unicode normalization, this can create a disparity between the hostname and path that are used for security checks made by the application and the hostname and path that are used for making a request. This disparity is frequently exploitable to bypass URL security logic.

Similar vulnerabilities can occur when Unicode hostnames are accepted as part of email addresses. There are Unicode characters whose KC normalization form contains ASCII characters with syntax-significance in a sequence of email addresses. These include “＠” (U+FF20, “Full-Width Commercial At”) and “;” (U+037E, “Greek Question Mark”), among others.

HostSplit vulnerabilities can occur in any case where security decisions are made based on a hostname. Because HostNames are security identifiers, misidentification of them can lead to vulnerabilities in many ways. RFC 6943 discusses this issue more generally.

HostSplit Example: Edge / IE and stealing OAuth tokens from Office 365

Although it is contrary to standards, many web sites issue HTTP response headers with a UTF-8 encoding. When performing HTTP redirects to Unicode hostnames such websites will include Unicode characters unescaped in the "Location" header that directs where the browser will redirect. Prior to being fixed as CVE-2019-0654, IE and Edge would convert Unicode characters encountered in the "Location" header of an HTTP redirect to their KC normalized form without checking whether any of these characters normalized to ASCII with syntax-significance in a URL. If Edge or IE received an HTTP redirect response directing the browser to a URL like "**https://www.evil.c℀.microsoft.com**" they would actually redirect to "**https://www.evil.ca/c.microsoft.com**".

A misdirected redirect of this type has significance where the OAuth protocol is concerned. The OAuth Authorization Code Flow Grant (RFC 6749) specifies that a user agent (browser) should pass a Client Identifier and Redirection URI to an Authorization Server which will authenticate the user and then redirect to the Redirection URI with an Authorization Code. Authorization Servers use an allow-list to determine which Redirection URI’s are valid for a given Client Identifier. This prevents malicious web sites from stealing authorization coded by sending an Authorization Code Flow request for an unaffiliated application.

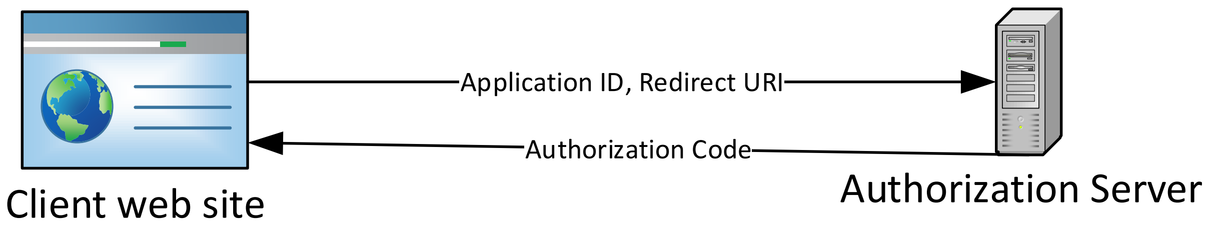

Figure 3 OAuth Authorization Code Flow

Many early OAuth implementations used an allow list logic of "allow any redirection URI whose hostname matches or is a subdomain of the specified domain", i.e. "*.office.com". The vulnerability in Edge and IE made it possible to bypass this type of allow list logic. A malicious web site could steal OAuth tokens from such an implementation by providing a URL like "**https://evil.c℀.office.com**". If this URL is checked against a pattern of "*.office.com" without first being normalized, it matches. However, when the Authorization Server attempts to redirect the user agent to this URL, Edge and IE would navigate to "**https://evil.ca/c.office.com**", causing the authorization code to be misdirected.

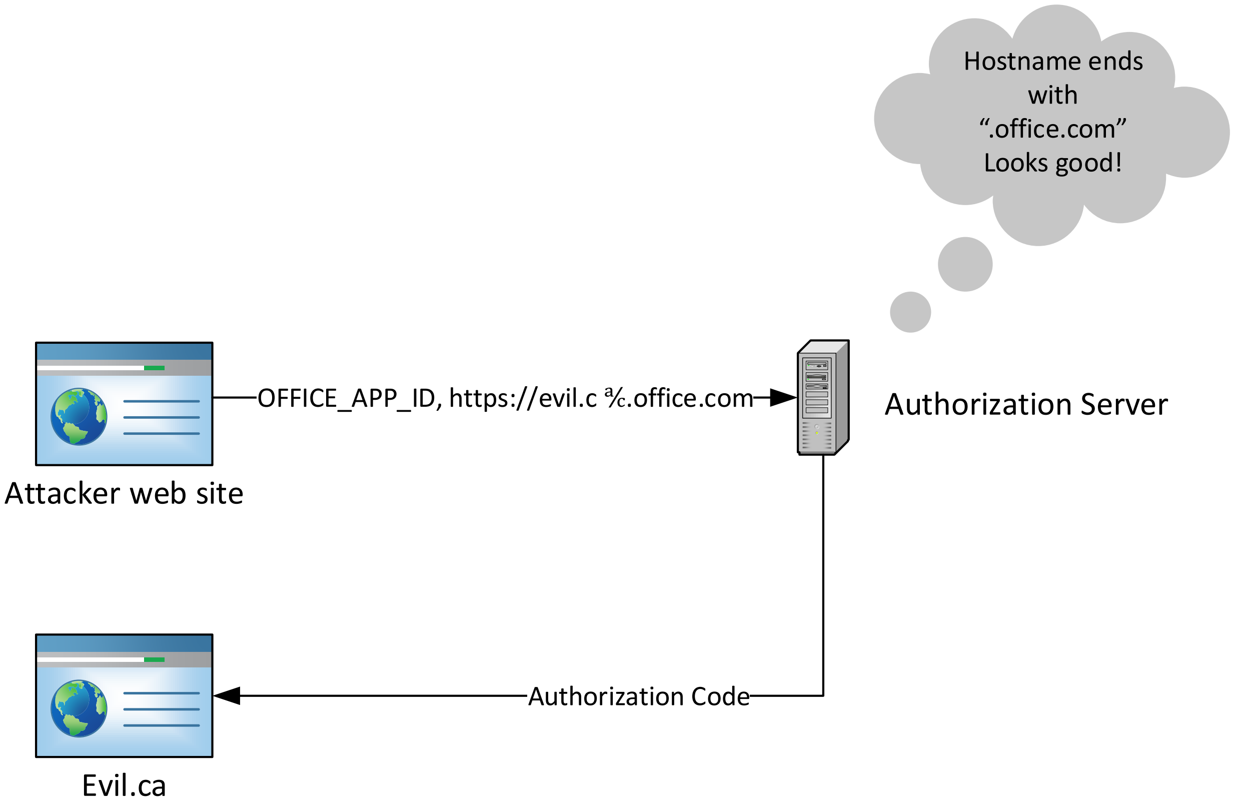
Hostname ends with ".office.com" Looks good!
OFFICE_APP_ID, https://evil.c ℀.office.com
Authorization Server
Attacker web site
Authorization Code
Evil.ca

Figure 4 Authorization Code Flow Subverted by HostSplit

Modern OAUTH implementations typically require the redirect URI allow list for an application to have a specific, fixed host. Such an allow list cannot be directly bypassed with HostSplit, but an indirect bypass is often possible by using redirects.

In 2014, a researcher named Wang Jing publicized an exploit pattern he named “Covert Redirect”. In his publication, he pointed out that sometimes a web application will have an open redirect vulnerability at a URL which is also a valid redirect URI for an OAUTH authenticator for that application. When this is the case, the OAUTH token can sometimes be captured by a site that is set as the target of the redirect.

The HostSplit vulnerability makes this attack pattern much easier. Many web applications contain redirect functionality that attempts to constrain the redirect to a specific domain or a subdomain of that domain. This redirect constraint can itself be bypassed using HostSplit.

Office 365 had a vulnerability of this type. Office.live.com was a valid redirect target for the Office OAUTH tokens and had a redirect function that attempted to allow redirects to dropbox.com or any subdomain of dropbox.com.

A URL of the following form would potentially allow the theft of an OAUTH token:

https://office.live.com/start/word.aspx?h4b=dropbox&eurl=https://evil.c℀.dropbox.com/wopi_edit/document1.docx&furl=https://www.dropbox.com/wopi_download/document1.docx&c4b=1

Because the redirect logic for this endpoint only checked that the redirect URL ended with “.dropbox.com” this redirect would be accepted, but if a user browsed to this page with Edge, the browser would follow the redirect to **https://evil.ca/c.dropbox.com** instead of **https://evil.c℀.dropbox.com**.

This specific vulnerability was mitigated by a separate bug in middleware used by Office 365 that caused UTF-8 encoded HTTP response headers to be UTF-8 double encoded.

The HostBond Vulnerability

The HostSplit vulnerability only works against software that implements either IDNA2003 or IDNA2008 + UTS46. IDNA2008 forbids all characters that the implementations of the other standards would normalize to ASCII characters with syntax significance in URL’s. However, there are two significant characters that IDNA2008 allows which IDNA2003 and IDNA2008+UTS46 do not.

IDNA2008 conditionally allows both the “Zero-Width Joiner” (U+200D) and “Zero-Width Non-Joiner” (U+200C) characters in Unicode hostnames. Because these characters are invisible, IDNA2003 forbids their use, because of the visual spoofing attacks they would enable. IDNA2008 allows these characters because of their significance in certain scripts. For example, in languages that use the Devanagari script, these joiner characters can change the way ligatures are rendered: “क्‌ष” vs “क्ष”. IDNA2008 only allows these joiner characters if the characters immediately preceding and following them would be rendered differently because of their presence.

IDNA2008 implementations generally apply this restriction correctly when converting U-labels to A-labels. A string like “micro” + zero-width joiner + ”soft” will have the zero-width joiner character discarded during normalization. The HostBond vulnerability relates to the way these characters are handled when an A-label is converted to a U-label.

RFC 3490 (IDNA) specifies that when an A-label is converted to a U-label, the result should then be converted back to an A-label and an error should be thrown if this A-label does not match the original input. Unfortunately, some IDNA implementations do not perform this round-trip check. Such implementations can be attacked by providing them an A-label containing a Punycode zero-width joiner or zero-width non-joiner.

The string “micro” + zero-width joiner + “soft.com”, converted to Punycode is:

xn--microsoft-469d.com

IDNA implementations that do not perform a round-trip check when converting A-labels to U-labels will decode this URL to the following U-label:

micro‍soft.com

(This hostname contains a zero-width joiner character, it’s just invisible.)

There are two problems with a hostname like this:

1. Since the zero-width joiner character is invisible, there’s no way for a user to distinguish this hostname from the ASCII hostname “Microsoft.com”
2. Because the zero-width joiner character is discarded when a U-label is converted to an A-label, systems which convert hostnames from ASCII to Unicode and then back to ASCII will cause this hostname to become the ASCII hostname “Microsoft.com”. This can allow software security checks to be bypassed.

HostBond Example: Impersonating sender email to Gmail

Gmail was vulnerable to the HostBond vulnerability.

Assume there is an email server at **email.somecloudhost.net** which an attacker wishes to impersonate. Assume also that the attacker can register a similar hostname, with the addition of a Punycode zero-width joiner between the “e” and “m” characters of the word “email”: **xn--email-xt3b.somecloudhost.net** . The attacker can then obtain a certificate for this domain using a service like Let’s Encrypt and set up DKIM, SPF, and DMARC.

The attacker then sets up an email server at this hostname and sends email from it using an address like admin@xn--email-xt3b.somecloudhost.net. When Gmail receives this email, it performs verification checks such as SPF and DKIM against the unaltered ASCII label. These succeed, because the attacker controls **xn--email-xt3b.somecloudhost.net** and has correctly established the necessary records. Gmail then converts the hostname to a U-label to display it to the user, so the user sees the email as having come from admin@e‍mail.somecloudhost.net (here the zero-width joiner is present, but invisible). If the recipient of the email replies, Gmail discards the zero-width joiner, and the email is sent to the non-spoofed email server.

How to test for HostSplit and HostBond

Testing for HostSplit

In cases where network traffic from potentially vulnerable software can be monitored, testing for HostSplit vulnerabilities is simple. It is sufficient to input a URL whose hostname contains a character whose KC-normalization form contains an ASCII character with syntax-significance in a URL and then check which hostname a DNS lookup is performed for.

A URL like **http://canada.c℀.bing.com** is an effective test case for scenarios of this type. Ideally, this URL should be rejected, but a DNS lookup for a Punycode subdomain of bing.com, while incorrect, should not be exploitable. A DNS lookup for “canada.ca” should be treated as a vulnerability.

In cases where network traffic cannot be directly monitored, a less direct test case can be used. If a tester controls two domains “A.com” and “B.com”, a URL like the following can be used as a test case:

http://a.com／X.b.com

In this string, the “／” character is the “Full-Width Solidus” character (U+FF0F).

For this test, a wildcard DNS record should be created that directs all subdomains of “b.com” to the same server. Software that is provided this URL and which is vulnerable to HostSplit will attempt to make a request to “a.com” for a file named “X.b.com”. Software that is not vulnerable will either not attempt to make a request or will instead attempt a request to a Punycode subdomain of “b.com”. Request logs from the servers pointed to by a.com and b.com can be monitored to determine which of these occurs.

Testing for HostBond

Testing for HostBond vulnerabilities is somewhat more complex. Software generally only converts A-labels to U-labels when displaying hostnames in a UI. Hence, testing for HostBond issues requires identifying all the places where an untrusted URL might be presented to a user.

Once a tester has identified functionality where an A-label is decoded into a U-label, the following test cases should be tried:

1. The hostname **xn--TC-m1t.com** should not be rendered as **TC.com** . This hostname contains a Punycode zero-width joiner, which should never be decoded to Unicode when the characters preceding and following it are both ASCII.
2. The hostname **x--orh.com** should not be rendered as **①.com** or **1.com**. This hostname contains a Punycode “Circled Digit One” (U+2460), whose KC normalization form is the ASCII numeral “1”.
3. The hostname **xn--ab-y4b.com** should not be rendered as **a;b.com**. This hostname contains a Punycode “Greek Question Mark” (U+037E), which is something of a special case, as it is often converted to ASCII by software that does not otherwise normalize Unicode characters.

To test email systems for HostBond vulnerabilities, a useful technique is to send email to a valid address while including an additional recipient with a Punycode hostname that contains characters that should not be decoded to Unicode. As an example, the address test@xn--bing-676a.com should not be rendered as “test@bing.com”.

Best Practices

Make hostname security decisions using ASCII

HostSplit vulnerabilities occur primarily when software receives an untrusted URL, makes security decisions regarding the URL, and then converts it to ASCII (possibly implicitly) in order to resolve it. The vulnerability occurs because of a disparity between how the hostname is parsed as Unicode and how it is parsed as ASCII. HostSplit vulnerabilities can be prevented by removing this disparity, either by only allowing users to input URL’s as ASCII or by converting URL’s to ASCII before making security decisions regarding them.

In cases where a user might provide a Unicode hostname, software must convert this hostname to an A-label before making any security decisions using it.

Use STD3ASCIIRules

The IDNA standard defines a flag “UseSTD3ASCIIRules” which, when set, indicates that the conversion of a U-label to an A-label should fail unless the resulting hostname consists only of the characters defined as legal in the “STD3” standard (RFC 1123). This standard allows alphanumeric characters, dashes, and periods. **This flag should be used whenever possible.**

Using this flag will entirely prevent HostSplit vulnerabilities, as they rely on the conversion from a U-label to an A-label introducing a syntax-significant character into a URL.

One caution should be taken: many legacy hostnames contain the ASCII underscore character. This is especially common in intranet hosts. Because the underscore character in a hostname is not legal under the STD3 rules, use of the UseSTD3ASCIIRules flag will make resources at these hostnames unreachable.

Due to this conflict, the use of underscores in hostnames should be deprecated.

Wrap vulnerable platform code

Many platform API’s perform unsafe conversions between U-labels and A-labels. When use of these API’s cannot be avoided, they should be wrapped in functions that ensure HostSplit and HostBond vulnerabilities are not possible.

For API’s that convert U-labels to A-labels, possibly to make a web request, a wrapper can be added to perform ToASCII independently of the API, with the UseSTD3ASCIIRules flag enabled. If the call to ToASCII fails, the wrapper function can transition to an exception flow before calling the API.

For API’s that convert A-labels to U-labels, a wrapper can be added to convert the A-label to a U-label and then back to an A-label. In cases where the resulting A-label does not match the input A-label, the wrapper flow can transition to an exception flow before calling the API.

Appendix I: List of related CVE’s and credits

CVE-2019-0654 *Microsoft Browser Spoofing Vulnerability*

CVE-2019-0657 *.NET Framework and Visual Studio Spoofing Vulnerability*

CVE-2019-9636 *Python, urlsplit does not handle NFKC normalization* - credit shared with Panayiotis Panayiotou

CVE-2019-10160 *Python, urlsplit NFKD normalization vulnerability in user:password@*

CVE-2019-2816 *Oracle Java SE/Java SE Embedded, “Normalize normalization”*

CVE-2019-12290 *LibIDN2, “Perform A-Label roundtrip for lookup functions by default”* - credit shared with Tim Ruehsen (GNU libidn), Florian Weimer (GNU glibc) and Nikos Mavrogiannopoulos (GnuTLS)

Special thanks to Tina Zhang-Powell of MSVR, for helping to coordinate these fixes.

Appendix II: List of potential URL-splitting Unicode characters

U+2100, ℀

U+2101, ℁

U+2105, ℅

U+2106, ℆

U+FF0F, ／

U+2047, ⁇

U+2048, ⁈

U+2049, ⁉

U+FE16, ︖

U+FE56, ﹖

U+FF1F, ？

U+FE5F, ﹟

U+FF03, ＃

U+FE6B, ﹫

U+FF20, ＠