Unicode Security

Software Vulnerability Testing Guide

(**DRAFT DOCUMENT** – this document is currently a preview in DRAFT form. Please contact me with corrections or feedback.)
Software Globalization provides a unique set of challenges for software engineers, and a rich attack surface for security researchers. This document attempts to shed light on some of the security issues exposed when developing Unicode-enabled software. Software engineers may find it useful in developing more secure code, and testers for finding security vulnerabilities.

July 2009
Casaba Security, LLC
www.casabasecurity.com

Chris Weber
Email: chris@casabasecurity.com
Blog: www.lookout.net

Table of Contents

1 Introduction

Unicode enables Globalized software, and is often discussed in terms of languages, strings and characters. Unicode itself is a single framework capable of representing all of the world's languages and scripts, past, present and future.

2 Unicode Background

The Unicode Standard provides a unique number for every character, enabling disparate computing systems to exchange and interpret text in the same way.

2.1 Brief History of Character Encodings

Early in computing history, it became widely clear that a standardized way to represent characters would provide many benefits. Around 1963, IBM standardized EBCDIC in its mainframes, about the same time that ASCII was standardized as a 7-bit character set. EBCDIC used an 8-bit encoding, and the unused eighth-bit in ASCII allowed OEM's to apply the extra bit for their proprietary purposes.

This allowed OEM's to ship computers and later PC's with customized character encodings specific to language or region. So computers could ship to Israel with a tweaked ASCII encoding set that supported Hebrew for example. The divergence in these customized character sets grew into a problem over time, as data interchange become error-prone if not impossible when computers didn't share the same character set.

In response to this growth, the International Organization for Standardization (ISO) began developing the ISO-8859 set of character encoding standards in the early 1980's. The ISO-8859 standards were aimed at providing a reliable system for data-interchange across computing systems. They provided support for many popular languages, but weren't designed for high-quality typography which needed symbols such as ligatures.

In the late 1980's Unicode was being designed, around the same time ISO recognized the need for a single character encoding framework, what would later come to be called the Universal Character Set (UCS), or ISO 10646. Version 1.0 of the Unicode standard was released in 1991 at almost the same time as UCS was made public. Since that time, Unicode has become the de facto character encoding model, and has worked closely with ISO and UCS to ensure compatibility and similar goals..

2.2 Brief Introduction to Unicode

The Unicode framework can presumably represent all of the worlds languages and scripts, past, present, and future. That's because the current version 5.1 of the Unicode Standard has space for over 1 million code points. A code point is a unique value within the Unicode code-space. A single code point can

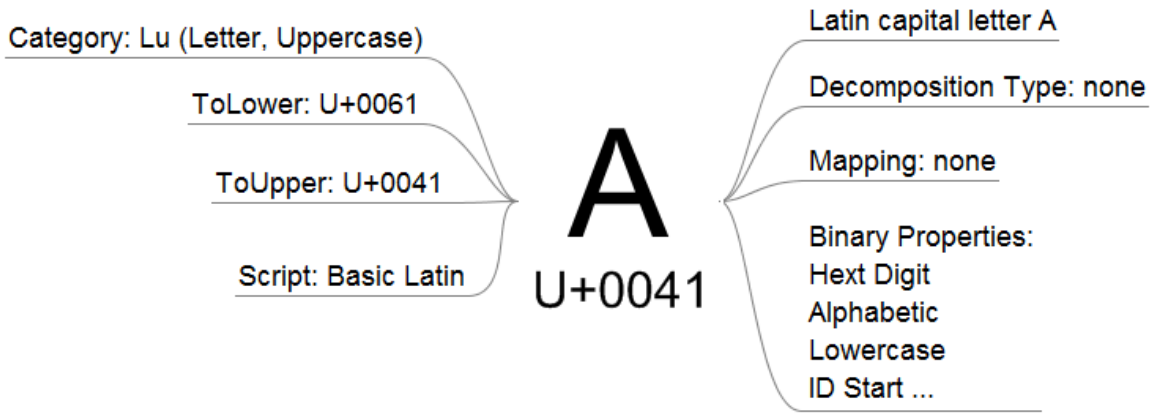
represent a letter, a special control character (e.g. carriage return), a symbol, or even some other abstract thing.

2.2.1 Code Points

A code point is a 21-bit scalar value in the current version of Unicode, and is represented using the following type of reference where NNNN would be a hex value:

U+NNNN

The valid range for Unicode code points is currently U+0000 to U+10FFFF. This range can be expanded in the future if the Unicode Standard changes. The following image illustrates some of the properties or metadata that accompany a given code point.



Code point U+0041 represents the Latin Capital Letter A. It's no coincidence that this maps directly to ASCII's value 0x41, as the Unicode Standard has always preserved the lower ASCII range to ensure widespread compatibility. Some interesting things to note here are the properties associated with this code point:

- Several categories are assigned including a general 'category' and a 'script' family.
- A 'lower case' mapping is defined.
- An 'upper case' mapping is defined.
- A 'normalization' mapping is defined.
- Binary properties are assigned.

This short list only represents some of the metadata attached to a code point, there can be much more information. In looking for security issues however, this short list provides a good starting point.

2.3 Character Encoding

A discussion of characters and strings can quickly dissolve into a soup of terminology, where many terms get mixed up and used inaccurately. This document will aim to avoid using all of the terminology, and may use some terms inaccurately according to the Unicode Consortium, with the goal of simplicity.

To put it simply, an encoding is the binary representation of some character. It's 'bits on the wire' or 'data at rest' in some encoding scheme. The Unicode Consortium has defined four character encoding forms, the Unicode Transformation Formats (UTF):

1. **UTF-7**
   Defined by RFC 2152.
2. **UTF-8**
   Each Unicode code point is assigned to an unsigned byte sequence of one to four bytes in length.
3. **UTF-16**
   Each Unicode code point is assigned to an unsigned sequence of 16 bits. There are exceptions to this rule, see the discussion of surrogate pairs below.
4. **UTF-32**
   Each Unicode code point is assigned to an unsigned sequence of 32 bits with the same numeric value as the code point.

Of these four, UTF-7 has been deprecated, UTF-8 is the most commonly used on the Web, and both UTF-16 and UTF-32 can be serialized in little or big endian format.

A **character encoding** as defined here means the actual bytes used to represent the data, or code point. So, a given code point `U+0041 LATIN CAPITAL LETTER A` can be encoded using the following bytes in each UTF form:

UTF Format	Byte sequence
UTF-8	< 41 >
UTF-16 Little Endian	< 41 00 >
UTF-16 Big Endian	< 00 41 >
UTF-32 Little Endian	< 41 00 00 00 >
UTF-32 Big Endian	< 00 00 00 41 >

The lower ASCII character set is preserved by UTF-8 up through U+007F. The following table gives another example, using `U+FEFF ZERO WIDTH NO-BREAK SPACE`, also known as the Unicode Byte Order Mark.

UTF Format	Byte sequence
UTF-8	< EF BB BF >
UTF-16 Little Endian	< FF FE >
UTF-16 Big Endian	< FE FF >

UTF-32 Little Endian	< FF FE 00 00 >
UTF-32 Big Endian	< 00 00 FE FF >

At this point UTF-8 uses three bytes to represent the code point. One may wonder at this point how a code point greater than U+FFFF would be represented in UTF-16. The answer lies in **surrogate pairs**, which use two double-byte sequences together. Consider the code point U+10FFFD PRIVATE USE CHARACTER-10FFFD in the following table.

UTF Format	Byte sequence
UTF-8	< F4 8F BF BD >
UTF-16 Little Endian	< FF DB > < FD DF >
UTF-16 Big Endian	< DB FF > < DF FD >
UTF-32 Little Endian	< FD FF 10 00 >
UTF-32 Big Endian	< 00 10 FF FD >

Surrogate pairs combine two pairs in the reserved code point range U+D800 to U+DFFF, to be capable of representing all of Unicode's code points in the 16 bit format. For this reason, UTF-16 is considered a **variable-width encoding** just as is UTF-8. UTF-32 however, is considered a **fixed-width encoding**.

2.4 Character Escape Sequences and Entity References

An alternative to encoding characters is representing them using a symbolic representation rather than a serialization of bytes. This is common in HTTP with URL-encoded data, and in HTML. In HTML, numerical character references (NCR) can be used in either a decimal or hexadecimal form that maps to a Unicode code point.

In fact, CSS (Cascading Style Sheets) and even Javascript use escape sequences, as do most programming languages. The details of each protocol's specification are outside the scope of this document, however examples will be used here for reference.

The following table lists the common escape sequences for U+0041 LATIN CAPITAL LETTER A.

UTF Format	Character Reference or Escape Sequence
URL	%41
NCR (decimal)	A
NCR (Hex)	A
CSS	\41 and \0041
Javascript	\x41 and \u0041
Other	\u0041

The following table gives another example, using U+FEFF ZERO WIDTH NO-BREAK SPACE, also known as the Unicode Byte Order Mark.

UTF Format	Character Reference or Escape Sequence
URL	%EF%BB%BF
NCR (decimal)	
NCR (Hex)	
CSS	\FEFF
Javascript	\xEF\xBB\xBF and \uFEFF
Other	\uFEFF

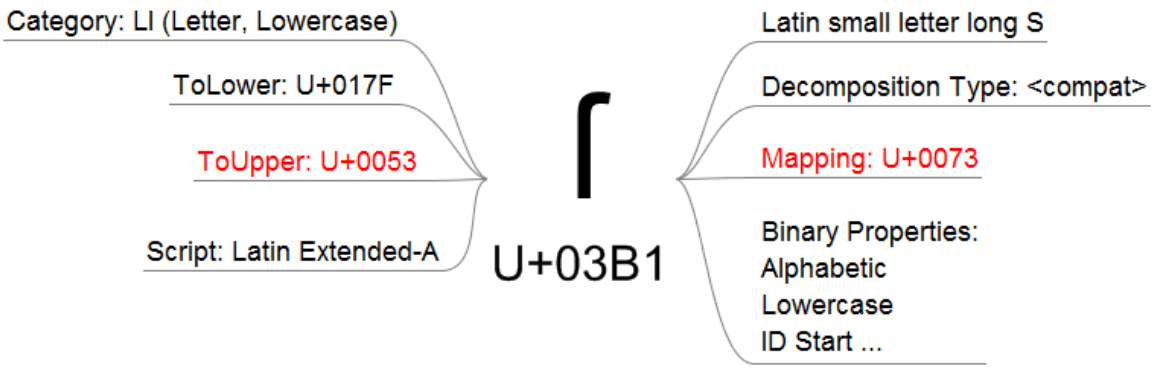
2.5 Focus Points for Security Testing

This security testing guide has been split into two general areas to aid readers in setting goals for a software security assessment. Where possible, data has also been provided to assist software engineers in developing more security software. Information such as how framework API's behave by default and when overridden is subject to change at any time.

Clearly, any protocol and standard can be subject to security vulnerabilities, examples include HTML, HTTP, TCP, DNS. Character encodings and the Unicode standard are also exposed to vulnerability. Sometimes vulnerabilities are related to a design-flaw in the standard, but more often they're related to implementation in practical use. Many of the phenomena discussed here are not vulnerabilities in the standard. Instead, the following categories can enable vulnerability in applications which are not built to prevent the relevant attacks:

- Visual Spoofing
- Best-fit mappings
- Charset transcodings and character mappings
- Normalization
- Canonicalization of overlong UTF-8
- Over-consumption
- Character substitution
- Character deletion
- Casing
- Buffer overflows
- Controlling Syntax
- Charset mismatches

Consider the following image as an example. In the case of U+017F LATIN SMALL LETTER LONG S, the **upper casing** and **normalization** operations transform the character into a completely different value. In some situations, this behavior could be exploited to create cross-site scripting or other attack scenarios.



The rest of this guide intends to explore each of these phenomena in more detail, as each relates to software vulnerability mitigation and testing.

3 Visual Spoofing

THE REST OF THIS SECTION IS BEING REMOVED UNTIL THE FULL RELEASE

4 Character and String Transformation Vulnerability

This section attempts to explore the various ways in which characters and strings can be transformed by software processes. Such transformations are not vulnerabilities necessarily, but could be exploited by clever attackers.

As an example, consider an attacker trying to inject a cross-site scripting attack into a Web-application with an input filter. The attacker finds that the application performs a lowercase operation on the input after filtering, and by injecting special characters they can exploit that behavior. That is, the string "script" is prevented by the filter, but the string "scrİpt" is allowed.

4.1 Round-trip conversions: a common pattern

In practice, Globalized software must be capable of handling many different character sets, and converting data between them. The process for supporting this requirement can generally look like the following:

1. Accept data from any character set, e.g. Unicode, shift_jis, ISO-8859-1.
2. Transform, or convert, data to Unicode for processing and storage.
3. Transform data to original or other character set for output and display.

In this pattern, Unicode is used as the broker. With support for such a large character repetoire, Unicode will often have a character mapping for both sides of this transaction. To illustrate this, consider the following Web application transaction.

1. An application end-user inputs their full name using characters encoded from the shift_jis character set.
2. Before storing in the database, the application converts the user-input to Unicode's UTF-8 format.
3. When visiting the Web page, the user's full name will be returned in UTF-8 format, unless other conditions cause the data to be returned in a different encoding encoding. Such conditions may be based on the Web application's configuration or the user's Web browser language and encoding settings. Under these types of conditions, the Web application will convert the data to the requested encoding.

The round-trip conversions illustrated here can lead to numerous issues that will be further discussed. While it serves as a good example, this isn't the only case where such issues can arise.

4.2 Best-Fit Mappings

The "best-fit" phenomena occurs when a character X gets transformed to an entirely different character Y. This can occur for reasons such as:

- A framework API transforms input to a different character encoding by default.
- Data is marshalled from a wide string type (multi-byte character representation) to a non-wide string (single-byte character representation).
- Character X in the source encoding doesn't exist in the destination encoding, so the software attempts to find a best match.

In general, best-fit mappings occur when characters are **transcoded between Unicode and another encoding**. It's often the case that the source encoding is Unicode and the destination is another charset such as shift_jis, however, it could happen in reverse as well. **Best-fit mappings are different than character set transcoding which is discussed in another section of this guide.**

Software vulnerabilities arise when best-fit mappings occur. To name a few:

- Best-fit mappings are not reversible, so data is irrevocably lost.
- Characters can be manipulated to bypass string handling filters, such as cross-site scripting (XSS) filters, WAF's, and IDS devices.
- Characters can be manipulated to abuse logic in software. Such as when the characters can be used to access files on the file system. In this case, a best-fit mapping to characters such as ../ or file:// could be damaging.

Shawn Steele describes the security issues well on his blog, it's a highly recommended short read for the level of coverage he provides regarding Microsoft's API's:

> *Best Fit in WideCharToMultiByte and System.Text.Encoding Should be Avoided. Windows and the .Net Framework have the concept of "best-fit" behavior for code pages and encodings. Best fit can be interesting, but often its not a good idea. In WideCharToMultiByte() this behavior is controlled by a WC_NO_BEST_FIT_CHARS flag. In .Net you can use the EncoderFallback to*

> *control whether or not to get Best Fit behavior. Unfortunately in both cases best fit is the default behavior. In Microsoft .Net 2.0 best fit is also slower.*

As a software engineer, it's important to understand the API's being used directly, and in some cases indirectly (by other processing on the stack). The following table of common library API's lists known behaviors:

Library	API	Best-fit default	Can override	Guidance
.NET 2.0	System.Text.Encoding	Yes	Yes	Specify EncoderReplacementFallback in the Encoding constructor.
.NET 3.0	System.Text.Encoding	Yes	Yes	Specify EncoderReplacementFallback in the Encoding constructor.
.NET 3.0	DllImport	Yes	Yes	To properly and more safely deal with this, you can use the MarshallAsAttribute class to specify a LPWStr type instead of a LPStr. [MarshalAs(UnmanagedType.LPWStr)]
Win32	WideCharToMultiByte	Yes	Yes	Set the WC_NO_BEST_FIT_CHARS flag.
Java	TBD			TBD
ICU	TBD			TBD

Another important note Shawn Steel tells us on his blog is that Microsoft does not intend to maintain the best-fit mappings. For these and other security reasons it's a good idea to avoid best-fit type of behavior.

The following table lists test cases to run from a black-box, external perspective. By interpreting the output/rendered data, a tester can determine if a best-fit conversion may be happening. Note that the mapping tables for best-fit conversions are numerous and large, leading to a nearly insurmountable number of permutations. To top it off, the best-fit behavior varies between vendors, making for an inconsistent playing field that does not lend well to automation. For this reason, focus here will be on data that is known to either normalize or best-fit. The table below is not comprehensive by any means, and is only being provided with the understanding that something is better than nothing.

Target char	Target code point	Test code point	Name
o	\u006F	\u2134	SCRIPT SMALL O
s	\u0073	\u017F	LATIN SMALL LETTER LONG S
I	\u0049	\u0131	LATIN SMALL LETTER DOTLESS I

K	\u004B	\u212A	KELVIN SIGN
a	\u0061	\u03B1	GREEK SMALL LETTER ALPHA
"	\u0022	\u02BA	MODIFIER LETTER DOUBLE PRIME
'	\u0027	\uFF07	FULLWIDTH APOSTROPHE
<	\u003C	\uFF1C	FULLWIDTH LESS-THAN SIGN
>	\u003E	\uFF1E	FULLWIDTH GREATER-THAN SIGN
:	\u003A	\u2236	RATIO

These test cases are largely derived from the public best-fit mappings provided by the Unicode Consortium. These are provided to software vendors but do not necessarily mean they were implemented as documented. In fact, any software vendor such as Microsoft, IBM, Oracle, can implement these mappings as they desire.

4.2.1 Guidance and Tooling

Avoid best-fit mappings if possible. Otherwise, perform best-fit mappings prior to security checks.

4.2.2 Tools

Casaba Security has a tool for runtime XSS testing of Web-applications that will inject special Unicode characters to identify transformations such as best-fit mappings. These transformations can be exploited for cross-site scripting XSS or other attacks.

THE REST OF THIS DOCUMENT IS BEING REMOVED UNTIL THE FULL RELEASE