# **Scott Stender**

scott@isecpartners.com

Black Hat Japan October 6, 2006



Information Security Partners, LLC iSECPartners.com

Introduction

# • Who are you?

- Founding Partner of Information Security Partners, LLC (iSEC Partners)
- Application security consultants and researchers

## • Why listen to this talk?

- Every application uses internationalization (whether you know it or not!)
- A great deal of research potential

### • Platforms

- Much of this talk will use Windows for examples
- Internationalization is a cross-platform concern!



#### • Introduction

#### • Background

- Internationalization Basics
- Platform Support
- The Internationalization "Stack"

#### • Historical Attacks

- Width calculation
- Encoding attacks

#### Current Attacks

- Conversion from Unicode
- Conversion to Unicode
- Encoding Attacks

#### • Tools

- I18NAttack
- Q&A

Background – Internationalization Basics

# • Internationalization Defined

- Provides support for *potential* use across multiple languages and localespecific preferences
- Most of this talk will focus on character manipulation

# Character Manipulation

- Text must be represented in 1s and 0s internal to the machine
- Many standards have emerged to encode text into a binary representation
- ASCII is a common example



Background – Internationalization Basics

	00	01	02	03	04	05	06	07	08	09	0A	<b>0</b> B	0C	0D	<b>0</b> E	0F
00	<u>NUL</u>	<u>STX</u>	<u>SOT</u>	<u>ETX</u>	<u>EOT</u>	<u>ENQ</u>	<u>ACK</u>	<u>BEL</u>	<u>BS</u>	<u>HT</u>	<u>LF</u>	<u>VT</u>	<u>FF</u>	<u>CR</u>	<u>SO</u>	<u>SI</u>
	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	0008	000C	000D	000E	000F
10	DLE	<u>DC1</u>	DC2	<u>DC3</u>	<u>DC4</u>	<u>NAK</u>	<u>SYN</u>	<u>ETB</u>	<u>CAN</u>	<u>EM</u>	<u>SUB</u>	<u>ESC</u>	<u>FS</u>	<u>GS</u>	<u>RS</u>	<u>US</u>
	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
20	<u>SP</u>	<u> </u>	<b>"</b>	#	\$	ଞ	&	<b>7</b>	(	)	*	+	,	-		/
	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C	002D	002E	002F
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
40	()	A	B	C	D	E	F	G	H	I	J	K	L	M	N	0
	0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	004A	004B	004C	004D	004E	004F
50	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	版 0057	X 0058	Y 0059	Z 005A	[ 005B	\ 005C	] 005D	へ 005E	005F
60	、	a	b	C	d	e	f	g	h	i	ј	k	1	m	n	0
	0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	006А	006B	006C	006D	006E	006F
70	р	q	r	S	t	u	V	W	X	У	Z	{		}	~	<u>DEL</u>
	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F

### **Binary Representations:**

APOSTROPHE = 0x27 = 0010 0111LATIN CAPITAL LETTER A = 0x41 = 0100 0001LATIN CAPITAL LETTER B = 0x42 = 0100 0010

Credit: http://www.microsoft.com/globaldev

Background – Internationalization Basics

# Code Pages

- Unicode
- Single-Byte: Most pages for European languages, ISO-8859-\*...
- Multi-Byte: Japanese (Shift-JIS), Chinese, Korean

### Encodings

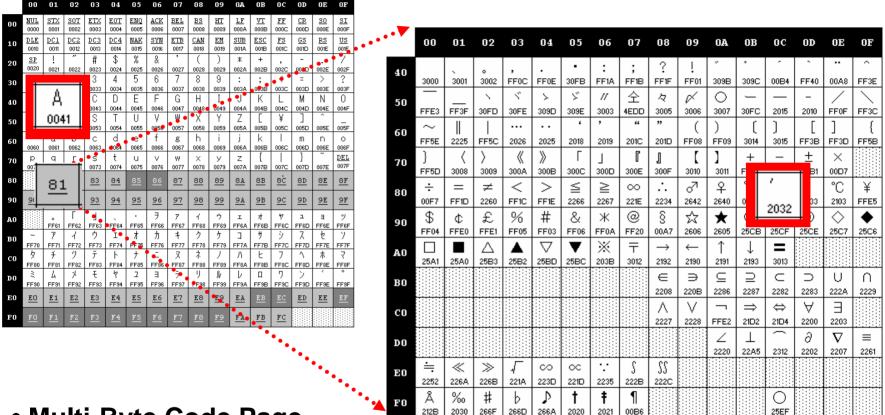
- EBCDIC, ASCII, UTF-7, UTF-8, UTF-16, UCS-2...

# • Encodings vs. Code Points

- Code pages describe sets of points
- Encodings translate those points to 1s and 0s
- Some standards don't require the distinction as much: ASCII
- Some are quite different: Unicode/UTF-8



Background – Internationalization Basics



### • Multi-Byte Code Page

0x41 = U+0041 = LATIN CAPITAL LETTER A

 $0x81 \ 0x8C = U+2032 = PRIME$ 

See http://www.microsoft.com/globaldev for others

Background – Internationalization Basics

### • Unicode

- Attempt to unify the world's characters into a single code page
- Current standards specify a 21-bit character space

# Unicode Encodings

- Though Unicode is often associated with 8 or 16-bit chars, these are just the most common encodings
- Many encodings available: UTF-32, UTF-16, UCS-2, UTF-8, UTF-7
- Many encodings, including UTF-16 and UTF-8 use a variable byte pattern

LATIN CAPITAL LETTER A = U+0041 = 0x41HALFWIDTH KATAKANA LETTER A = U+FF71 = 0xEF 0xBD 0xB1



Background – Platform Support

# • OS provides core of support

- Windows core text is UTF-16 encoded
- Linux Standard Base requires UTF-8 string support

# • Support isn't just from the OS

- Programming language
- Virtual machines
- Application only

### • This offers a unique attack surface

- Cross-OS, Language, Application Class, and Implementation
- A great place to start is with standards that stipulate I18N support
- In short, this hits almost every application out there



Background – Platform Support

### Character Manipulation Support

- Everything required to support cross-code page data
- Everything required to support encodings

## Unicode often used as the canonical representation

- This makes sense given that it is the unified code page

### • Each platform uses similar patterns for converstion

- Code page source destination can be inferred
- Parameters of conversion there are hard decisions to make
- Core data support source and destination locations
- Let's look at some examples...



Background – Platform Support

# • MultiByteToWideChar – Convert to Unicode

- CodePage can use default (CP\_ACP) which will vary by system
- Note all of the length specifiers!

#### int MultiByteToWideChar(

UINT CodePage,	code page					
DWORD dwFlags,	character-type options					
LPCSTR lpMultiByteStr,	/ string to m	ap				
int cbMultiByte,	/ number of b	ytes in string				
LPWSTR lpWideCharStr,	/ wide-charac	ter buffer				
int cchWideChar	/ size of buf	fer				
);						



Background – Platform Support

#### WideCharToMultiByte – Convert from Unicode ${}^{\bullet}$

- dwFlags modifies conversion properties
  - WC\_NO\_BEST\_FIT\_CHARS is your friend!
- lpDefaultChar allows you to specify error character

#### int WideCharToMultiByte(

UINT CodePage,

DWORD dwFlags,

- // code page
- // performance and mapping flags
- LPCWSTR lpWideCharStr, // wide-character string
- int cchWideChar, // number of chars in string
- LPSTR lpMultiByteStr, // buffer for new string
- int cbMultiByte, // size of buffer
- LPCSTR lpDefaultChar, // default for unmappable chars
- LPBOOL lpUsedDefaultChar // set when default char used

#### );

Background – Platform Support

- Almost every platform has support for internationalization
  - Results depend on Unicode standard supported by platform
- Newer platforms tend to play nicer with Unicode
  - .Net & Java use native Unicode encodings, though they can convert to others
- Great, I use one of those!
  - Your application still depends on the internationalization support of underlying OS, servers they interact with, etc.
  - You still have to worry these attacks



Background – The Internationalization Stack

- Every application has internationalization dependencies
  - Development platform
  - External libraries
  - Operating System
  - Application Server
  - Database Server collations!
  - Clients



Background – The Internationalization Stack

# • Each level acts as a potential "internationalization boundary"

- Your app may get it right, but the next layer up or down might not!

### • The Default Code Page

- Remember CP\_ACP?
- Change system and user locales
- Ever tried to test your app on other languages?
- How about throughout the stack?



Background – The Internationalization Stack

# • Web applications

- Code page can be set on both HTTP request and response

```
user=test&pass=test
```



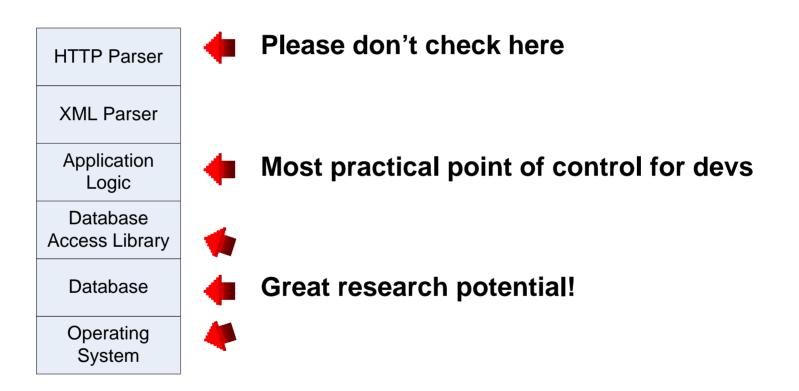
Background – The Internationalization Stack

# • Web applications

- Code page is set on first line of every XML document

```
<?xml version="1.0" encoding="utf-8" ?>
<TestXML>
<Data>
This is test data
</Data>
</TestXML>
```

Background – The Internationalization Stack





#### • Introduction

#### • Background

- Internationalization Basics
- Platform Support
- The Internationalization "Stack"

#### Historical Attacks

- Width calculation
- Encoding attacks

#### Current Attacks

- Conversion from Unicode
- Conversion to Unicode
- Encoding Attacks

#### • Tools

- I18NAttack
- Q&A

Historical Attacks – Width Calculation

### • Security and Internationalization has seen some attention...

- Chalk these up as "lesson learned," for the most part
- Attack Pattern Incorrect Width Calculation
  - Conversion functions
  - Count of bytes vs. Count of characters
    - sizeof(array) vs. sizeof(array)/sizeof(array[0])
  - Compile-time function specifiers (lstr\*, tchars) affect sizes

### • Buffer Overflow

- Destination buffer assumed to be 1 byte/character
- Reported destination buffer is count of bytes rather than count of characters



Historical Attacks – Encoding Attacks

• Attack Pattern - non-minimal UTF-8 encodings

### Consider an HTTP Server

- I would like to request a file called blah.html off a web server

### • Legitimate requests have simple encodings:

- http://.../web/index.html
- http://.../web/../../blah
- http://.../web/%2E%2E%2F%2E%2E%2F/blah
- It is easy enough to look for .. / %2E%2E and %2F
- Unusual encodings can bypass validation routines:
  - %C0%AE is a non-minimal UTF-8 encoding for %2E
  - http://.../web/%C0%AE%C0%AE%C0%AF%C0%AE%C0%AF/blah



#### • Introduction

#### • Background

- Internationalization Basics
- Platform Support
- The Internationalization "Stack"

#### • Historical Attacks

- Width calculation
- Encoding attacks

#### Current Attacks

- Conversion from Unicode
- Conversion to Unicode
- Encoding Attacks

#### • Tools

- I18NAttack
- Q&A

Current Attacks – Conversion from Unicode

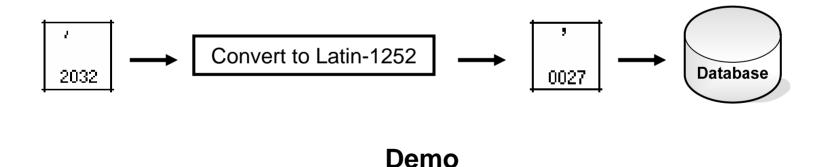
• Scenario – Validation is performed on input, changed to locale-specific text

- Attack Class "Use Best-Fit Equivalents"
  - Unicode's character space is much larger than any locale-specific code page
  - Results in a many-to-one mapping for many characters
  - Code-page specific
  - Big reason why WC\_NO\_BEST\_FIT\_CHARS should always be specified



Current Attacks – Conversion from Unicode

- Sneaking an apostrophe in...
  - U+2032 = PRIME
  - Converted to Latin-1252 it is 0x27 Apostrophe
  - Same thing happens for quotation marks, numbers, letters, etc.
  - Latin-1 isn't the only code page, have you tried your other supported languages as well?





Current Attacks – Conversion to Unicode

- Scenario Validation is performed on input, later converted to Unicode
- Attack Class "Eating Characters"
  - Many languages rely on "escape characters" to cleanse data
  - Validation routines will often identify and escape as appropriate
  - Eating one of the characters will counteract this validation routine

### • Use a multi-byte encoding scheme

- A converter will identify lead byte, and interpret trail bytes accordingly
- Just send up a lead byte by itself...



Current Attacks – Conversion to Unicode

- Eating a SQL quotation character
  - Using Shift-JIS MBCS Japanese Code Page
  - Interpret as Unicode

```
0x82 0x60 = FULLWIDTH LATIN CAPITAL LETTER A
0x82 0x27 = Not mapped, converts to default char (?)
0x82 0x27 0x27 = Not mapped plus apostrophe (?')
```

### • Consider a database...

– Table users requires support for names with an apostrophe select \* from users where name = 'O''Henry'

- Submit a last name that ends in 0x82

```
select * from users where name = 'O''Henry?
```

- Submit a last name that ends in 0x82' or 1=1--

select \* from users where name = 'O''Henry?' or 1=1-



Current Attacks – Encoding Attacks

- Scenario Validation is performed on input, changed to an alternate encoding
- Attack Class "Foiling Canonicalization"
  - The IIS4 vuln required that %C0%AE be interpreted as 0x2E or simply '.'
  - One easy way to fix disallow non-minimal encoding support
  - Indeed, the Unicode standard was changed
- What to do with the illegal characters
  - Causing an error is not usually acceptible in widely distributed applications
  - What happens if every unusual character caused a database to skip a transaction?
  - Most UTF-8 parsers today choose to omit such characters rather than fault



Current Attacks – Encoding Attacks

# • Legitimate requests have simple encodings:

- http://.../web/index.html
- http://.../web/../../blah
- http://.../web/%2E%2E%2F%2E%2E%2F/blah
- ..easy enough to look for .. / %2E%2E and %2F

## • Unexpected encodings can bypass validation routines:

- %C0%AE is a non-minimal UTF-8 encoding for %2E
- http://.../web/.%C0%AE./.%C0%AE./blah
- ../ or direct variants not found in input, so passed to file access routine
- File parser converts .%C0AE./.%C0AE./ to UTF-16 (as NtCreateFile requires)
- Non-minimal encodings dropped ../../ remains

### Demo



Current Attacks – Encoding Attacks

# • Attack Class – "Mistaken Identity"

- We have been spoiled by the most common Unicode encodings
- Unicode is just a set of code points, encoding is up to the parser
- UTF-8, UTF-16, and UCS-2 all resemble ASCII

### • UTF-7

- 7-bit encoding designed to work with ASCII-only SMTP
- Most printable ASCII characters are encoded directly
- Everything else is encoded as UTF-16, modified base64 encoded, and wrapped with + and –
- Sneak "garbage" data past validators
  - Most interesting characters exist in ASCII ', ", <, >, =...
  - Validation routines often take advantage of the ASCII resemblance
  - Many encodings can easily bypass this approach
  - ASCII, EBCDIC, UTF7..



Current Attacks – Bonus!

### • Timestamp Attacks

- Is 10-06-06 October 6, 2006 or June 6, 2010?
- Your ticket expiration check might want to know!

# • Sorting Attacks

- Which comes first, apple or aardvark? How about in Danish?
- Your search & validation routine might want to know!

### • What is a proper decimal separator?

- Your CSV-based storage routine might want to know



#### • Introduction

#### • Background

- Internationalization Basics
- Platform Support
- The Internationalization "Stack"

#### • Historical Attacks

- Width calculation
- Encoding attacks

#### Current Attacks

- Conversion from Unicode
- Conversion to Unicode
- Encoding Attacks

#### • Tools

- I18NAttack
- Q&A

Tools – I18NAttack

# • Background

- Testing equivalence characters, "eaters," alternate encodings is time consuming!
- Goal is to provide a security-focused collection of characters and encodings that often trip up input validation routines
- Using it is always going to be transport-dependent, but here is a tool to get you started...

# • I18NAttack

- HTTP POST/GET Parameter Fuzzer
- Reference implementation for nasty character database
- Will identify and fuzz problem characters across equivalents, unusual encodings, etc.
- Use to bypass poor input validation

### Demo



# Q&A

### **Scott Stender**

scott@isecpartners.com

