UNICODE と
サニタイジング回避テクニック

2006 年 12 月 06 日

Ver. 1.2

目　次

1　UNICODE とサニタイジング回避テクニック

1.1 UNICODE とサニタイジング回避テクニックとは(本文書の目的)

本文書は、2004 年 10 月 30 日の「まっちゃ 139 勉強会」(※)のはせがわようすけ氏の「Unicode とセキュリティ」について、より詳細に研究した内容を記述する。

また、本文書の目的は、不正行為の助長ではなく、このようなセキュリティ上の問題が存在することを周知徹底させることが目的であり、その結果コンピュータソフトウェアの更なる発展を期待するものである。

(※) まっちゃ 139 勉強会 : http://matcha139.hiemalis.org/~isamik/benkyokai.html#02

1.2 UNICODE を使ったサニタイジング回避テクニックの本質

「1.1 UNICODE とサニタイジング回避テクニックとは」で説明した参考資料の通りであるが、少しだけ本質論を記述する。

UNICODE を使ったサニタイジング回避テクニックは、セキュリティ対策としてのサニタイジングが

1. サニタイジング処理 (※1)
2. 処理実行

の二段階で行われる処理の流れの中で、1 のサニタイジング処理を空間の広い UNICODE 上で行い、次段階の 2 の実際の処理作業時には ANSI(※2)に変換され、1 のサニタイジングされていない(サニタイジング処理を迂回した)データ(メタキャラクタ)によってセキュリティ侵害を起こす、というセキュリティ侵害行為テクニックである。

結論、UNICODE プログラムと ANSI プログラムが並存している環境で発現する危険性がある。

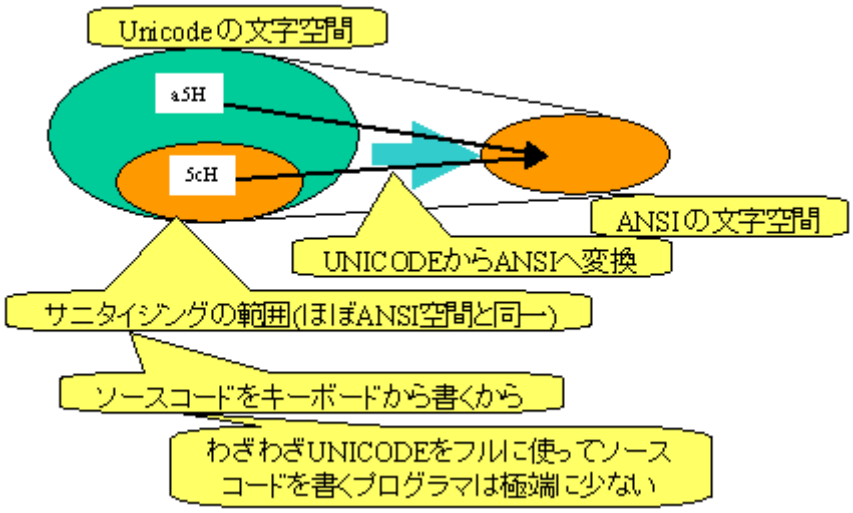

図1.2-1 : UNICODE **を使ったサニタイジング回避テクニックの概要** 1

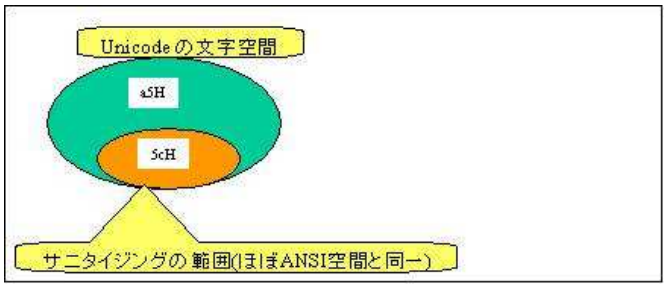

図1.2-2 : UNICODE **を使ったサニタイジング回避テクニックの概要** 2

(プログラムがサニタイジングするがサニタイジング範囲は限定的)

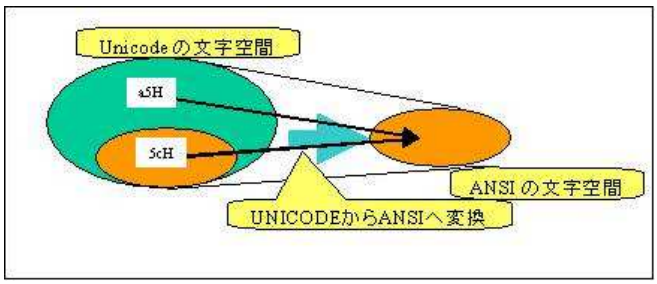

図1.2-3 : UNICODE **を使ったサニタイジング回避テクニックの概要** 3

(プログラムが呼び出した関数内部で ANSI **へ変換しているような場合、図** 1.2-2**のサニタイジングを迂回される)**

(※1) サニタイジング処理 : 本文書では "サニタイジング処理" と "エスケープ処理" はほぼ同義である。

(※2) ANSI：ここでは、UNCODE 以前の文字コードをさしている。つまり、日本国内の場合、ASCII+SJIS または ASCII+EUC-JP などの文字コードのことである。

1.3 UNICODE を使ったサニタイジング回避テクニックへの対策

■ **サニタイジング処理で対策する 1**

サニタイジング処理部で対策するには、サニタイジング処理前にデータを ANSI 文字コードに対しての正規化を行ってしまえばよい。

つまり、対象データを一度 ANSI に変換してから、もう一度 UNICODE へ変換しなおすという処理を行うことである。

当然だが、ANSI 化されることで、国際化プログラムではなくなるが、元々UNICODE ではないモジュールを使っているため、国際化プログラムではなくなるということはデメリットではないだろう。

- VisualBASIC6 では、strconv() 関数によって、UNICODE と ANSI との変換が可能である。
  (srtconv(データ,vbFromUnicode)→strconv(データ,vbUnicode))
- VBScript 環境では strConv() 関数はないが、VB の ActiveX-DLL を作ってもらい、それを使用するといいだろう。
  (一応、http://www.cc.rim.or.jp/~sanaki/text/free/free50.htm の Moji_Chk.dll の reg_unicode()を用意した)
- C++の場合、Win32API の WideCharToMultiByte()/MultiByteToWideChar()を使って、ANSI 文字コードに対して正規化した UNICODE 文字列に変換すればよい。

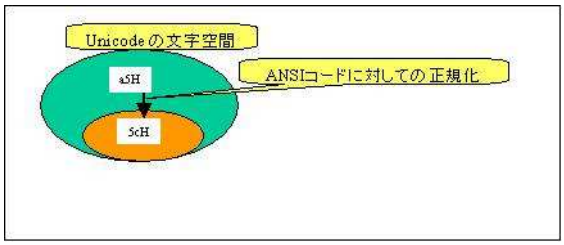

図1.3-1：UNICODE を使ったサニタイジング回避テクニックの対策 1

(サニタイジング前に ANSI に対しての正規化をしてしまう)

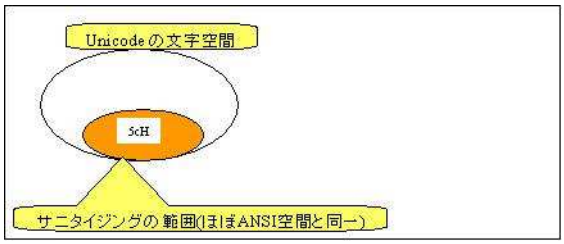

図1.3-2 : UNICODE **を使ったサニタイジング回避テクニックの対策** 2

(普通にサニタイジング処理を実施)

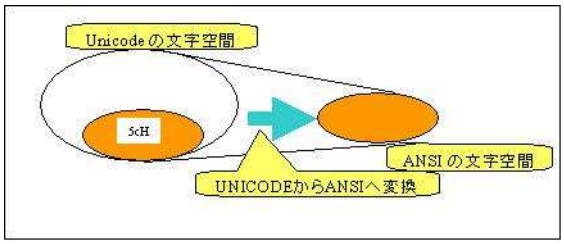

図1.3-3 : UNICODE **を使ったサニタイジング回避テクニックの対策** 3

(普通に内部的に ANSI **変換するメソッドを呼び出す)**

注意点は、上記の変換処理(UNICODE→ANSI→UNICODE**)がコンパイラの最適化によって省かれないように注意する必要がある(※)。**

■ **サニタイジング処理で対策する** 2

サニタイジング処理時に、ANSI **変換によって同一化する** UNICODE **文字も考慮してサニタイジング処理を行う。**

この方法は理想的ではあるが、いくつか問題点がある。

- UNICODE **から** ANSI **へ変換する関数は複数種類あるため、それぞれに同一化される文字が異なる(**2 UNICODE **を使ったサニタイジング回避テクニックの影響を参照)。**
  利用するモジュールがどのような変換によって UNICODE **から** ANSI **へ変換されるかをモジュールを利用する側の開発者が把握しておく必要があり、これは現実的ではない。**
- **そもそも、利用するモジュール側で** ANSI **化されるため、**UNICODE **という広い文字空間を維持する必要性が薄い。**

■ **処理側で対策をする**

処理側では、常に UNICODE **で処理を行うように記述する。**

(より正確には与えられた文字コードのままで処理を完結すること)

具体的には文字列を扱う場合 char **型ではなく、**wchar **型を使うようにする。**

- Windows-VisualC++**の場合、**tchar.h **と** tchar **型を使い、プリプロセッサの宣言を「**_MBCS**」から「**_UNICODE**」に書き直す。**

  (Windows9x **系での動作に支障がでる可能性に注意)**

図1.3-4 : VC6++SP6 **のプリプロセッサの設定画面**(Win32ConspleApplication & MFC)

デフォルトでは「_MBCS**」がついてビルドされる**

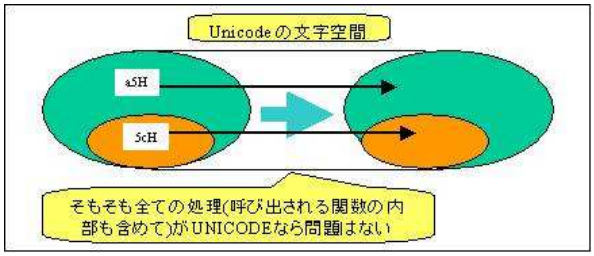

図1.3-5 : 呼び出されるメソッド内部も含めて、そもそも全ての処理が UNICODE **で行われていれば問題はない**

(※) **参考 : 良いニュースと悪いニュース**

http://www.microsoft.com/japan/msdn/columns/secure/secure10102002.asp

(ここではコンパイラの最適化によって、メモリクリアのコードが抜け落ちる可能性を指摘している)

2　UNICODE を使ったサニタイジング回避テクニックの影響

2.1 UNICODE を使ったサニタイジング回避テクニックの影響

全てのプログラムが UNCODE 化される近未来においては、このセキュテリィ問題は収束するだろうと予想できる。
しかし、一部のソフトウェアのみが UNICODE に対応している現状においては、「UNICODE を使ったサニタイジング回避テクニック」は広範囲で潜在的に存在しているものと推定される。

例えば、WindowsNT 系列の OS は内部的に UNICODE でありながら ANSI なプログラムを受け入れるような下位互換性を保持している。
また、Java 言語は UNICODE で記述されている。また Perl 言語、Python 言語、Ruby 言語なども内部的に UNICODE 化している。
これらの言語で開発したプログラムやスクリプトが、ANSI なモジュールを利用する時、UNICODE を使ったサニタイジング回避テクニックについて注意する必要がある。

2.2 WindowsNT 系の場合

WindowsNT では、内部的に UNICODE を採用している。採用している UNICODE のコード体系は、16bit 固定であり UCS-2 をそのまま使っている(UTF-16)(一部に変則あり)。

WindowsNT の API には、WideCharToMultiByte()/MultiByteToWideChar()という関数があり、UNICODE(WideChar)と ANSI(MultiByte)の相互変換を行うことができる。

この関数を用いて、どのように重複するか確認した。(「6.1 WindowsNT 系列での UNICODE 変換リストのプログラムについて」)

その結果は以下である。

0x21[!] (u00a1)	0x2d[-] (u00ad)
0x31[1] (u00b9)	0x32[2] (u00b2)
0x33[3] (u00b3)	0x41[A] (u00c0 ～ u00c6)
0x43[C] (u00c7)	0x44[D] (u00d0)
0x45[E] (u00c8 ～ u00cb)	0x49[I] (u00cc ～ u00cf)
0x4e[N] (u00d1)	0x4f[O] (u00d2 ～ u00d8)
0x52[R] (u00ae)	0x54[T] (u00de)
0x55[U] (u00d9 ～ u00dc)	0x59[Y] (u00dd)
0x5c[\] (u00a5)	0x61[a] (u00aa, u00e0 ～ u00e6)
0x64[d] (u00f0)	0x65[e] (u00e8 ～ u00eb)
0x69[i] (u00ec ～ u00ef)	0x6e[n] (u00f1)
0x6f[o] (u00ba, u00f2 ～ u00f8)	0x73[s] (u00df)
0x74[t] (u00fe)	0x75[u] (u00f9 ～ u00fc)
0x79[y] (u00fd, u00ff)	0x7c[¦] (u00a6)

0x3f [?]は、変換できない場合に変換されるデフォルトの文字であるため、除外した。

2.3 Java の場合

Java 言語は内部的に UNICODE を採用している。

Java 言語では、<String>.getBytes(<ANSI コード名>)という String クラスのメソッドを使うことで、UNICODE → ANSI への変換が可能である。

このメソッドを用いて、どのように重複するか確認した。(「6.2 Java での UNICODE 変換リストのプログラムについて」)

その結果は以下である。

0x5c[\] (u00a5)	0x7e[~] (u203e)

3　UNICODE を使ったサニタイジング回避テクニックの具体例

3.1 ANSI-C で書かれた Windows プログラム

WindowsNT 系の OS は内部的に UNICODE を採用している。
しかし、WindowsNT 系の OS で動作するプログラム(アプリケーション)までもが UNICODE 化されているとは限らない。

WindowsNT 系の OS で動作するプログラムを ANSI で記述した場合、以降で実験している各種の UNICODE を使ったサニタイジング回避テクニックでのセキュリティ問題が潜在的にあるということになる。

つまり、

- WindowsNT 系の OS で動作する ANSI なプログラムでは、ファイルパスを含む文字列データは ANSI コード(0x00 を終端とする char 型配列)に縮退しているため、このようなプログラムから NTFS 系ファイルシステムへアクセスした場合、UNICODE を使ったサニタイジング回避テクニックによって、「バックスラッシュ(0x5c)」が無害化されない危険性がある。
- WindowsNT 系の OS で動作する ANSI なプログラムでは、外部コマンド呼び出し用のコマンド文字列を含む文字列データは ANSI コード(0x00 を終端とする char 型配列)に縮退しているため、このようなプログラムから(CMD.exe 経由で)外部コマンドを呼び出した場合、UNICODE を使ったサニタイジング回避テクニックによってサニタイジングが回避される危険性がある。
- WindowsNT 系の OS で動作する ANSI なプログラムでは、データベースへ問い合わせを行う SQL 文字列を含む文字列データは ANSI コード(0x00 を終端とする char 型配列)に縮退しているため、このようなプログラムから(PostgreSQL や mySQL などの「バックスラッシュ(0x5c)」でエスケープできる)データベースに対して SQL 実行を依頼した場合、UNICODE を使ったサニタイジング回避テクニックによってサニタイジングが回避される危険性がある。
- その他にもモジュールや関数への入力データの書式によって、UNICODE を使ったサニタイジング回避テクニックによってサニタイジングが回避される危険性がある。

当然、ANSI な Windows プログラム上でサニタイジング処理を実施していれば、(サニタイジング時に文字列データは ANSI コードに縮退しているため)本文書のサニタイジング回避テクニックによる危険性は発生しない。

しかし、ANSI プログラムの呼び出し元が、UNICODE プログラムであり、かつその UNICODE プログラムである呼び出し元でサニタイジング処理を実施している場合、UNICODE を使ったサ

ニタイジング回避テクニックによってサニタイジングが回避される危険性がある。

一般的に、Windows 上での UNICODE プログラムと ANSI プログラムでは、ソースコード・レベルで互換性がない。
つまり、UNICODE 系では文字列データとして wchar 配列を用い、ANSI 系では char 配列を用いる。さらに、呼び出す各種関数について、異なっている。(一般的に UNICODE 系は「W」の付いている関数を呼び出し、ANSI 系では「A」のついている関数を呼び出す)。

また、Windows9x 系の OS での UNICODE 系の関数にバグが多かったという過去の事例もあり、多くのプログラマは、Windows9x 系の OS での動作を非動作にしてまで UNICODE プログラムを作成するようなこともないと思われる。

一方で、Windows プログラムを作成する際、ANSI な Windows9x 系と、UNICODE な WindowsNT 系でのソースコードの互換性を取るために、tchar.h を使うことがしばしばある。
このように tchar.h を使ってソースコードを作成した場合、Windows9x 系では ANSI プログラムとして動作し、WindowsNT 系では UNICODE プログラムとして動作させることができる。
しかし、UNICODE 系になるのか ANSI 系になるのかは、プログラム実行時に決定されることではなく、コンパイル時に決定される。
つまり、プログラムのバイナリ形式は異なるのである。
(よく、Win9x 系と WinNT 系でインストーラが異なるのはこのためである。また Setup.exe は同一でありながらそこから呼び出される msi ファイルが UNICODE 版と ANSI 版に分けられている場合もある)

図3.1-1 : tchar.h **を使ったプログラムの** Windows95 SP1 **での実行例**

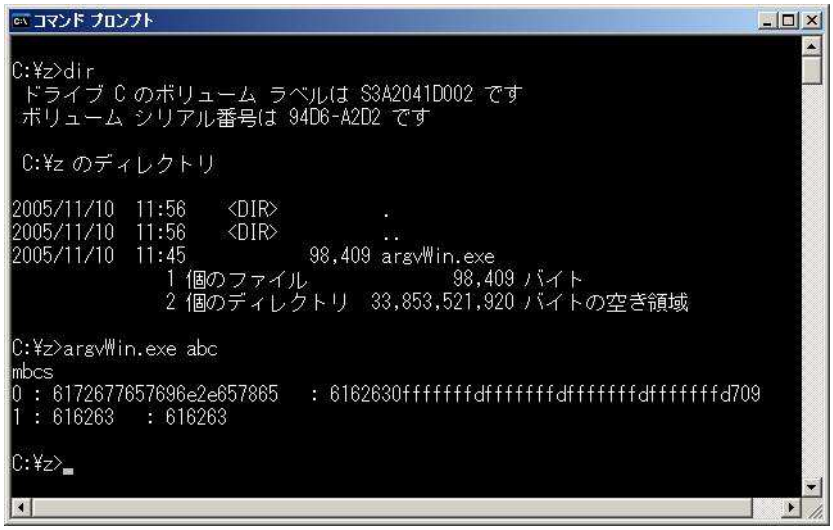

図3.1-2 : tchar.h **を使ったプログラムの** WindowsXP SP2 **での実行例(図** 3.1-1**と同一バイナリ)**

バイナリが同一だと、WinNT **系でも** ANSI **なプログラムとして動作する**

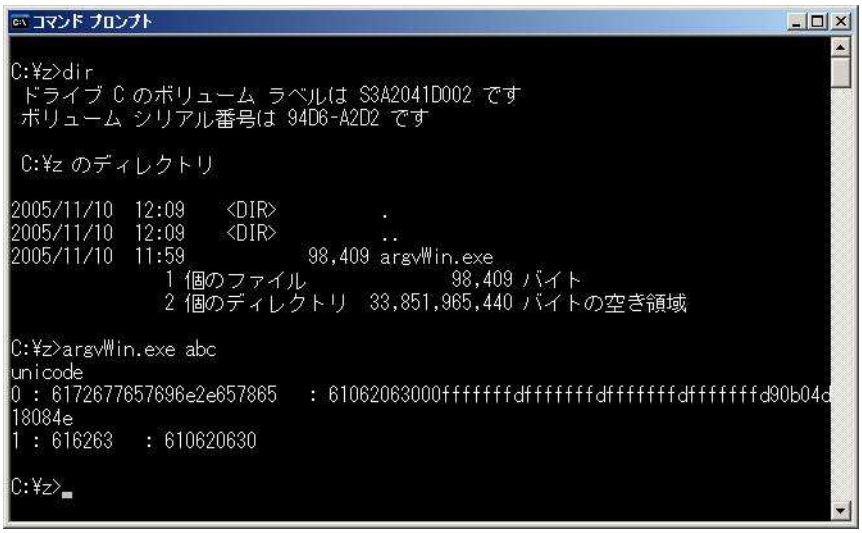

図3.1-3 : tchar.h **を使ったプログラムの** WindowsXP SP2 **での実行例**(_UNICODE **オプション付ビルド**)

また、この tchar.h **のデフォルトであるが、**MS-VisualC++6.0SP6 **では、「**_MBCS**」である。つまり、**ANSI **である。**
全てではないが、MS-VisualC++6.0SP6 **のウィザードによって生成されるプロジェクトでの「ビルド構成」の多くは、コンパイル時に「**_MBCS**」を指定している。**
この設定の変更は、プリプロセッサの定義を変更することである。(「メニュー」→「プロジェクト」→「設定」→「C/C++**」→「プリプロセッサの定義」)**

以上のことから推測すると、Windows9x **系でも** WindowsNT **系でも動作する同一バイナリの** Windows **プログラムは、「**_MBCS**」オプションでコンパイルされているのではないか思われる。つまり、このような** Windows **プログラムを** WindowsNT **系で動作させた場合、**UNICODE **を使ったサニタイジング回避テクニックによってサニタイジングが回避される危険性を誘発する恐れがある。**

また、COM **についても、内部的に** UNICODE **処理していない場合、この** COM **を呼び出した場合(呼び出し元は内部的に** UNICODE **で処理されるスクリプトなどが多いと思われる)、**UNICODE **を使ったサニタイジング回避テクニックによってサニタイジングが回避される危険性を誘発する恐れがある。**

以下は、「6.4Variant **型の引数を** ANSI **で受け取るメソッドと** Unicode(Binary)**で受け取るメソッドのある** COM(MS-VC++6.0SP6)**」を「**6.5**「**6.4**」の** COM **を呼び出すテストスクリプト」から呼び出し結果である。**

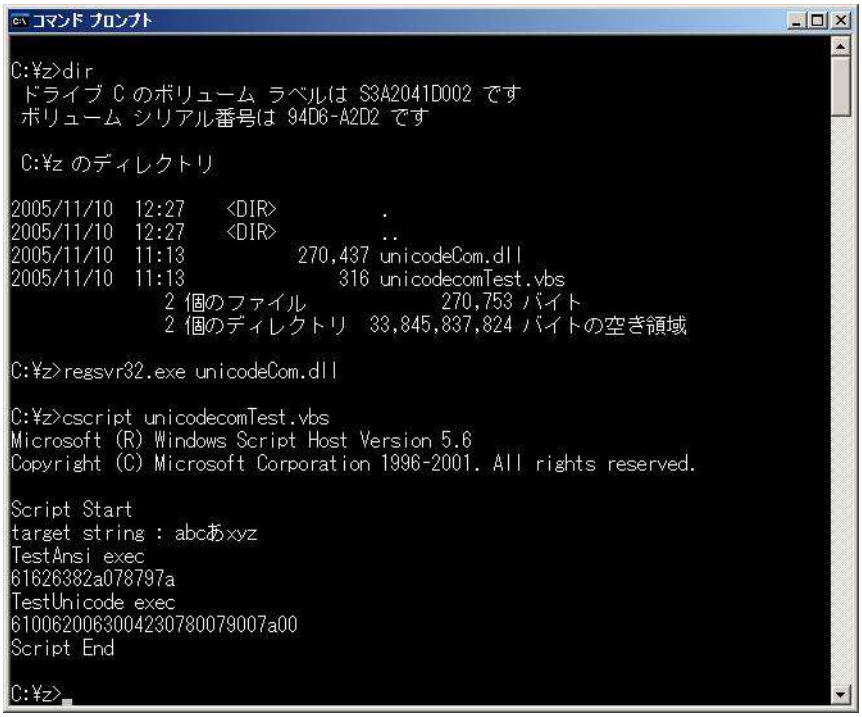

図3.1-4：「6.4」「6.5」で紹介したプログラムの実行結果

このように、例え COM のメソッドであったとしても、コーディングのしやすさ(wchar 配列のコーディング例よりも char 配列についてのコーディング指南書や教科書が多い)から、モジュールのメソッドの内部で、ANSI への置換処理を行っている可能性もある。
実際に筆者の一人が公開しているソフトウェア(http://www.cc.rim.or.jp/~sanaki/text/free/sanaki-8.stm 以下)の MS-VC++6 .0 で記述された COM プログラムは、文字列処理として ANSI を採用している。

結論として、Windows 上でプログラムをする場合、極力 wchar 型でコーディングすることで UNICODE を使ったサニタイジング回避テクニックを抑えることが可能である。また、char 型でのコーディング時には使用者(モジュールを使用しているスクリプターなど)にその旨通知し、スクリプターは必要に応じて、ANSI への正規化処理などの UNICODE を使ったサニタイジング回避テクニック対策を実装させるようにして使わせることでも UNICODE を使ったサニタイジング回避テクニックを抑えることが可能である。

3.2 Windows ファイルシステム

Windows ファイルシステムでは、「バックスラッシュ(0x5c)」がパスのデリミタとして機能している。
つまり、UNICODE上で「バックスラッシュ(u005c)」をチェックしていたとしても「円記号(u00a5)」をチェックしていないということになり、ANSIモジュールによって、「円記号(u00a5)」が0x5cのANSI文字(バックスラッシュ)へと置き換えられてしまう。

3．2.1 Scripting.FileSystemObject オブジェクト(WSH5.6)

WSH/ASP/VB などでファイルアクセスする時、使われるオブジェクトである。
ソースコードは、「6.8 VB の Open ステートメントと Scripting.SystemFileObject オブジェクトを使ったファイル読み出しプログラム (VB6SP6)」を参照。
今回は､(WSH や ASP などと比較して)バイナリ処理が簡単な VisualBASIC6.0SP6 を選択した。

このプログラムでは、「UNICODE Bug」チェックをつけると、「バックスラッシュ(0x5c)」を円記号(u00a5)に変換する。

「a(u00a5)test.txt」というファイルを作る流れが図 3．2.1-1～図 3．2.1-3である。
図より、「u00a5」がディレクトリデリミタの「バックスラッシュ(0x5c)」に置換されていないため、Scripting.FileSystemObject を使う限りにおいて、UNICODE を使ったサニタイジング回避テクニックのセキュリティ上の問題はない。

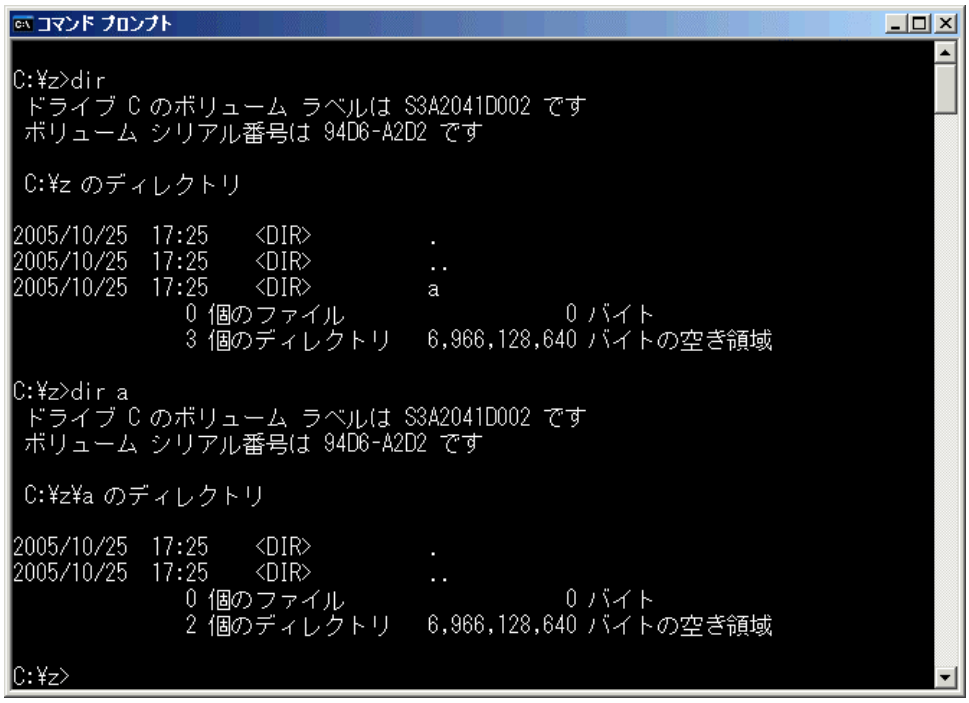

図3．2.1-1：テストプログラム実行前

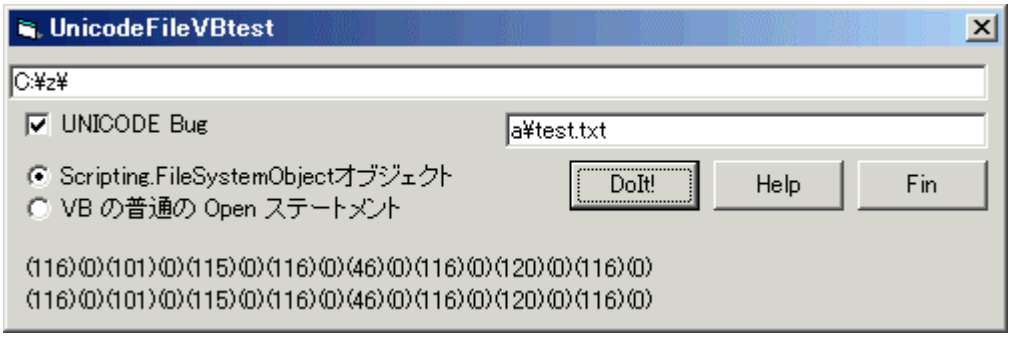

図3．2.1-2：ScriptingFileSystem **オブジェクトで「**a(u00a5)test.txt**」というファイルを作る**

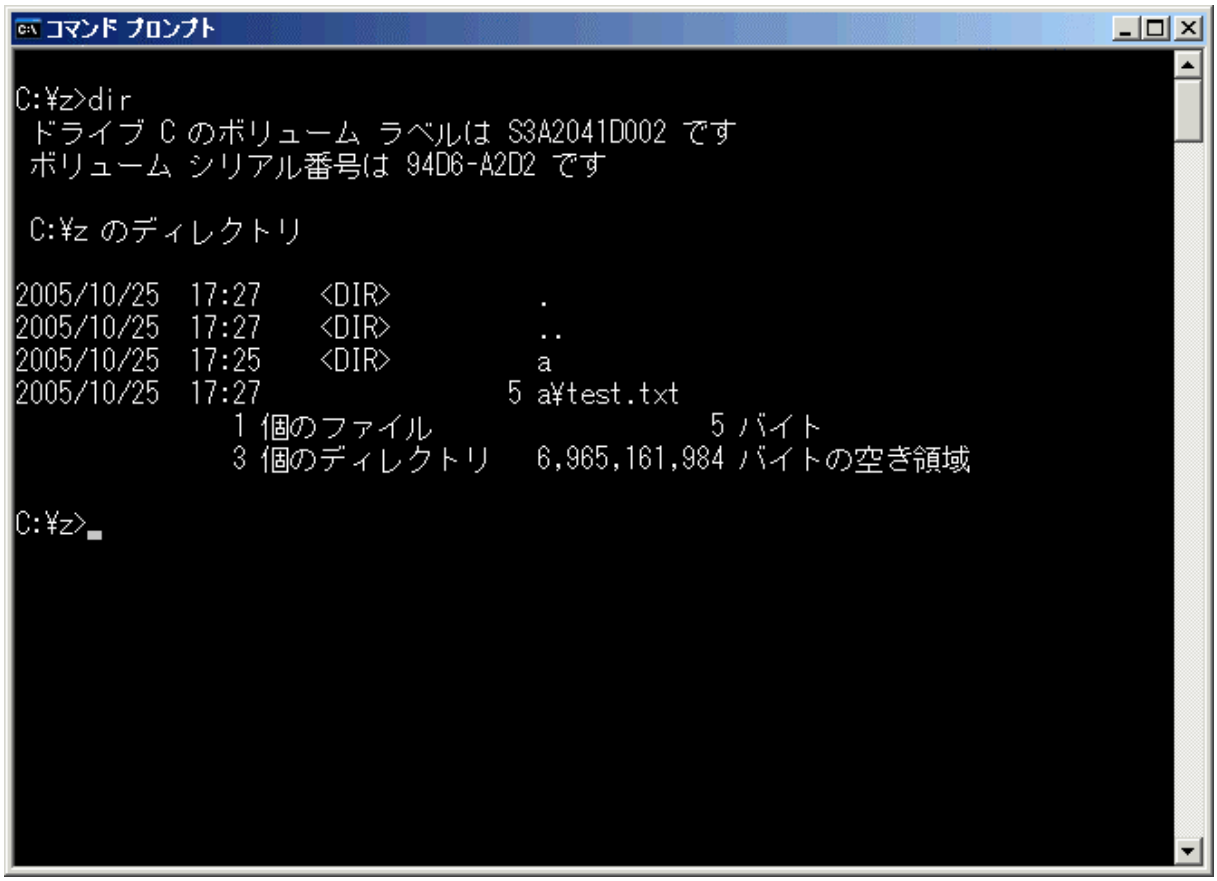

図3．2.1-3：図 3．2.1-2の結果(「u00a5」がファイル名となり、問題はない)

3．2.2　open ステートメント(VisualBASIC6.0SP6)

VisualBASIC では、旧来からファイル・アクセス時に用いている open ステートメントがある。

この open ステートメントを使った場合に、UNICODE を使ったサニタイジング回避テクニックの脆弱性があるかどうか確かめてみた。図 3．2.2-1～図 3．2.2-2がそれである。
ソースコードは、「6.8 VB の Open ステートメントと Scripting.SystemFileObject オブジェクトを使ったファイル読み出しプログラム (VB6SP6)」を参照。
この図より「a(u00a5)test1.txt」というファイルを作成しようとしたのにも関わらず、open ステートメントでは、「u00a5」をディレクトリ・デリミタの「バックスラッシュ(0x5c)」に変換してしまっていることが分かる。
つまり、「バックスラッシュ」のサニタイジング処理を回避される可能性がある。

open ステートメントを呼び出す前にディレクトリ・トラバーサル問題対策のために「..(0x5c)」などサニタイジング処理を行っていたとしても、「..(u00a5)」を与えることで、任意のファイルへのアクセスが可能となる。

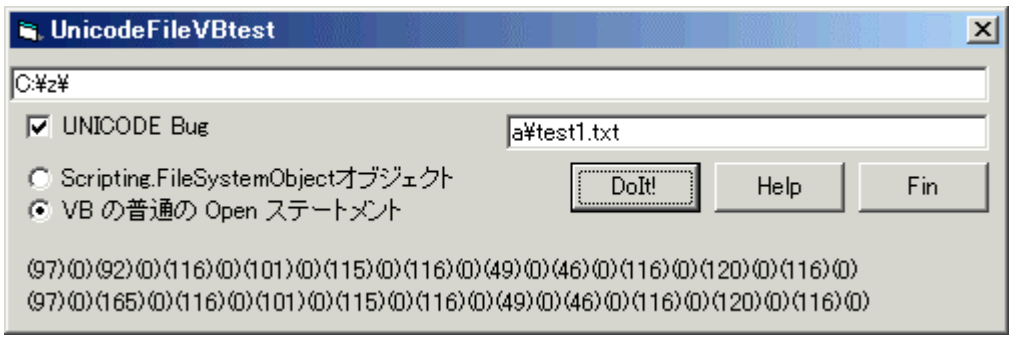

図3．2.2-1：今度は、open ステートメントにチェックする

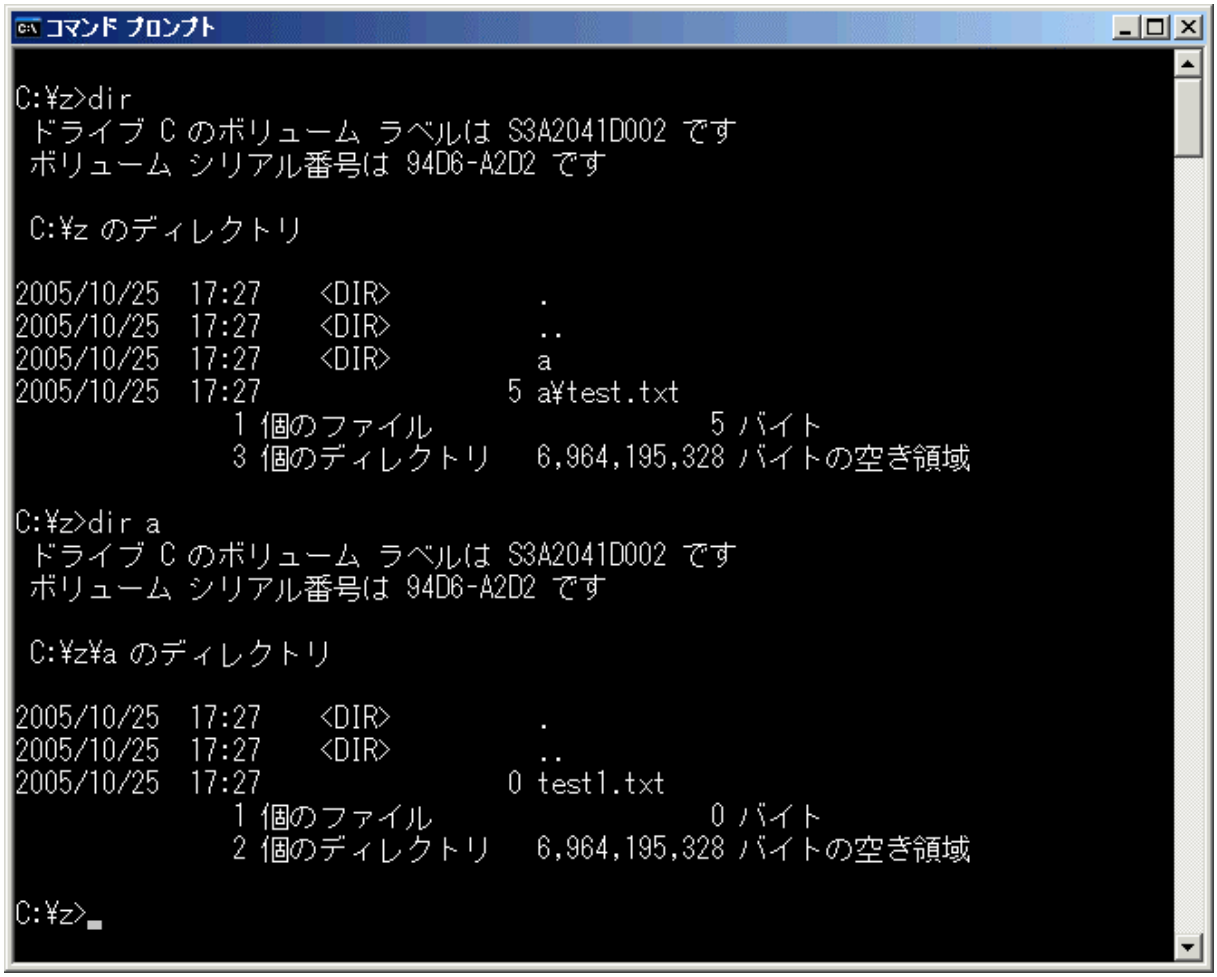

図3．2.2-2：図 3．2.2-1の結果

3．2.3　Windows 上での Java

Java 言語は内部処理は UNICODE で行っている。
Java 言語のファイルシステムへのアクセスでは File クラスを使うのが基本的な方法である。
実際にテストをしてみた結果、Windows 上での Java プログラムは、ファイル名を UNICDE で扱っており、UNICODE を使ったサニタイジング回避テクニックの危険性は発生しない。

図 3．2.3-1のように、c:¥java¥a(u00a5)test.txt というファイルが、「c:¥java¥a¥test.txt」としてファイルが生成されていないことから、Windows 上での Java のファイルアクセスにおいては、UNICODE を使ったサニタイジング回避テクニックの危険性は発現しないものと思われる。

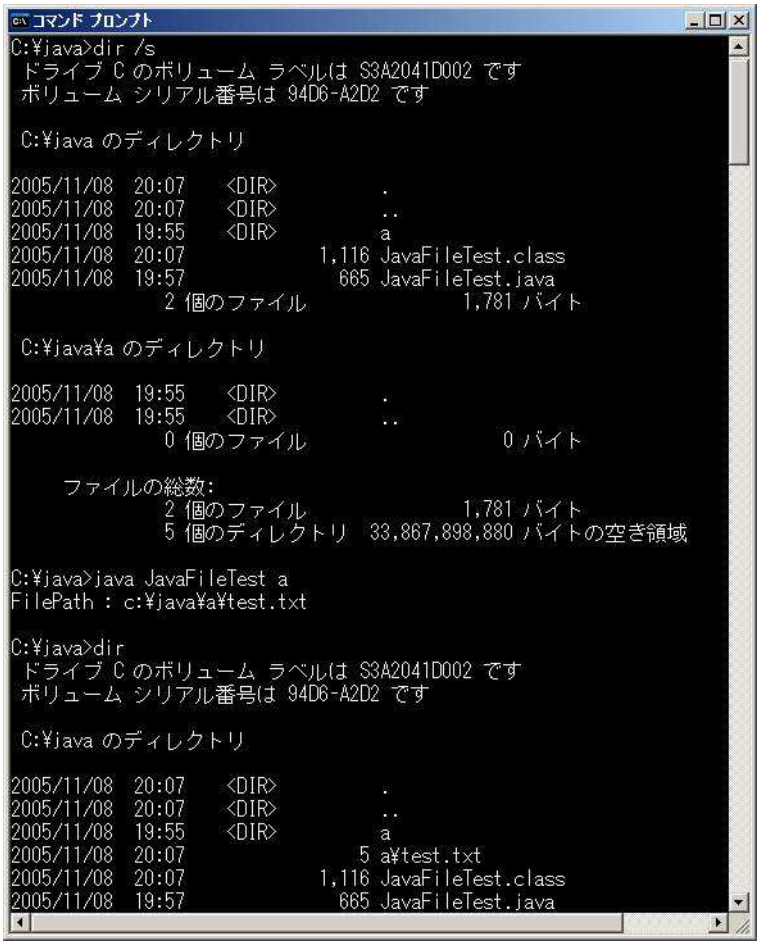

図3．2.3-1 : Java **プログラムと** UNICODE **ファイルパスと** Windows

3．2.4　Windows 上での Python2.4.2

現在、スクリプト言語全盛の時代であるが、Perl,Python,Ruby **は** Windows **上でも動作する。これらのファイルアクセスについても調査した。**
Python **の最新版は、**UNICODE **化されており、**Windows **上で動作する** Python **を用いてテストした。(実行環境は** Python 2.4.2 (#67, Sep 28 2005, 12:41:11) [MSC v.1310 32 bit (Intel)] on win32)
ソースコードは、「6.7 Python **で書かれたファイル読み出しプログラム」である。**

結果は、図 3．2.4-2**である。**
このことから、ファイルパス中の「u00a5**」記号が、ディレクトリデリミタの「バックスラッシュ(**0x5c**)」に置換されることはないということになる。つまり、**Python **で書かれたスクリプトでのファイルパスによる** UNICODE **を使ったサニタイジング回避テクニックの影響はない。ということになる。**

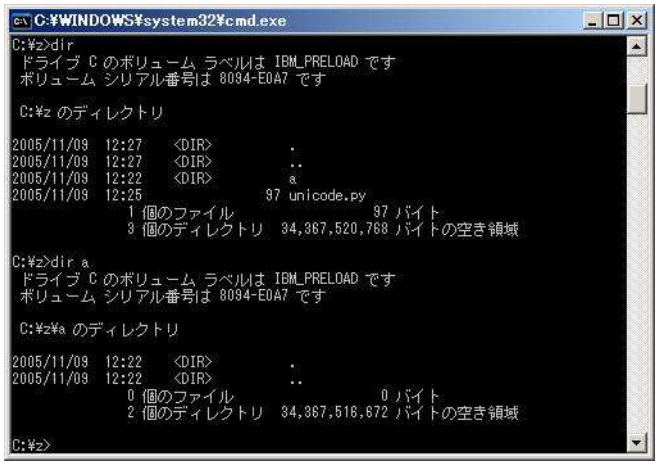

図3．2.4-1：テスト前のディレクトリ状態「c:¥z **以下」「**c:¥z¥a **以下」には** unicode.py **以外はない**

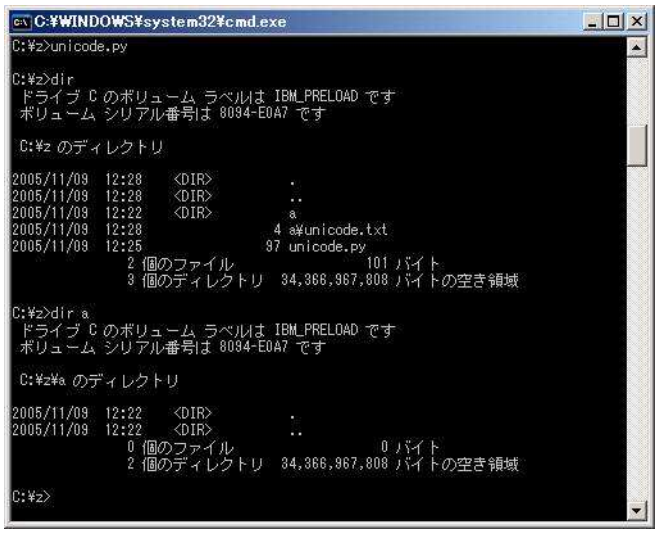

図3．2.4-2：テスト実行後。「c:¥z」直下にファイルができていることから、UNICODE バグの問題はない

3.3 Windows の OS コマンド呼び出しについて

Windows では、「|(0x7c)」が UNICODE から ANSI への変換で同一化される文字になっている。「|(0x7c)」はパイプである。しかし、Windows シェル(CMD.EXE)は基本的に UNICODE で処理されるため、問題ないはずである。また、Windows 上のモジュールはシェルを経由することなく外部コマンドを実行する場合が多いため、そもそも「OS コマンドインジェクション」の危険性は少ない(引数の強制指定という脅威の可能性は否定しない)。しかし、ANSI な system() 関数を用いている場合、「|(0x7c)」のサニタイジングに注意する必要がある。

3．3.1 WindowsScriptingHost 5.6 の WScript.Shell オブジェクトの Run()メソッド

WSH で外部コマンドを実行する場合、WScript.Shell オブジェクトの Run メソッドを使うことが多い。
Run メソッド自体は、cmd.exe を呼び出していないため「OS コマンドインジェクション」とは無縁の存在ではある(図 3．3.1-1と図 3．3.1-2)。
(引数の強制指定という脅威の可能性は否定しない)

しかし、Run メソッドから cmd.exe を呼び出して外部コマンドを使えば、「|(0x7c)」(パイプ)などのシェルの機能を利用することができる。

そこで、Run メソッドは内部的にどのような文字コードを使っているか調査したのが、図 3．3.1-3と図 3．3.1-4である。
図 3．3.1-3と図 3．3.1-4で結果が異なっていることから、u00A6 の記号は、シェルのパイプ(0x7c)を意味するコードに変換されないことが分かる。
よって、WScript.Shell の Run メソッドは UNICODE を使ったサニタイジング回避テクニックに対して安全である。

ちなみに外部コマンドとして呼び出しているプログラムは、a.exe,b.exe 共に「6.11 外部コマンド呼び出しによって呼び出されるプログラム」を VisualC++6.0SP6 でコンパイルしたものをファイル名を「a.exe」および「b.exe」として保存したものである。

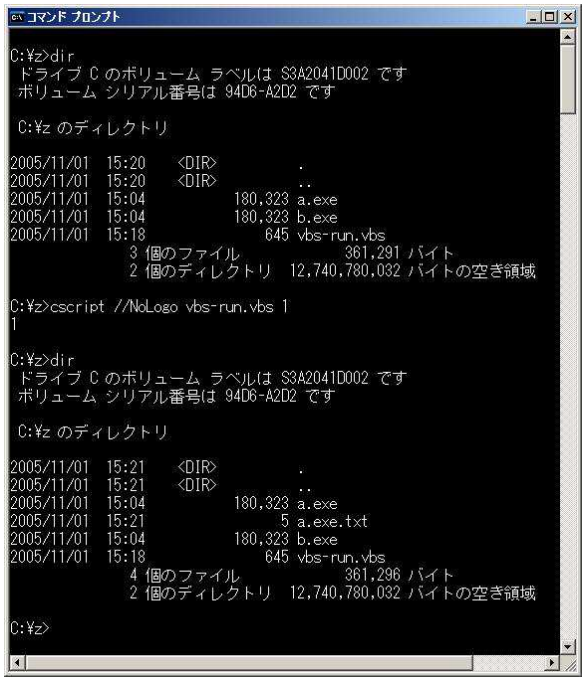

図3．3.1-1：「a.exe|b.exe**」の結果**

b.exe.txt **がないことから「|」以下はコマンドとして実行されていない**

(cmd.exe **を明示的に呼び出す必要がある**)

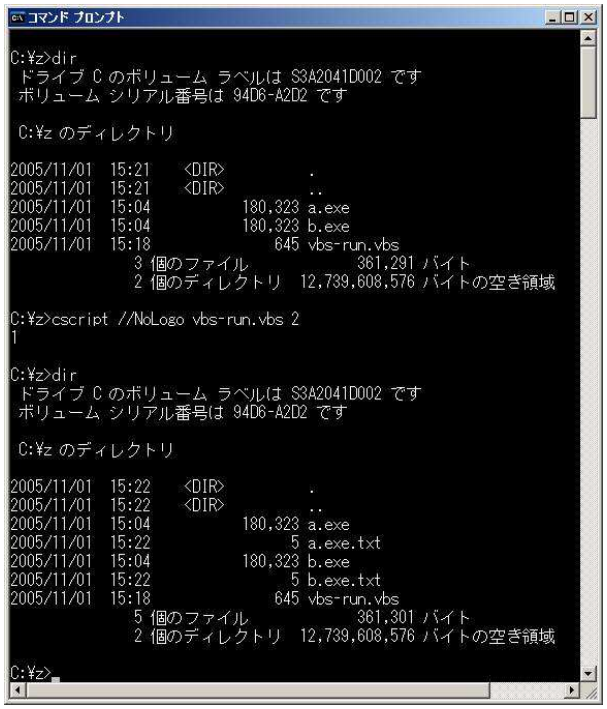

図3．3.1-2：「cmd.exe /c a.exe|b.exe**」の結果**

(cmd.exe **を明示的に呼び出すことで「パイプ」機能を利用できる**)

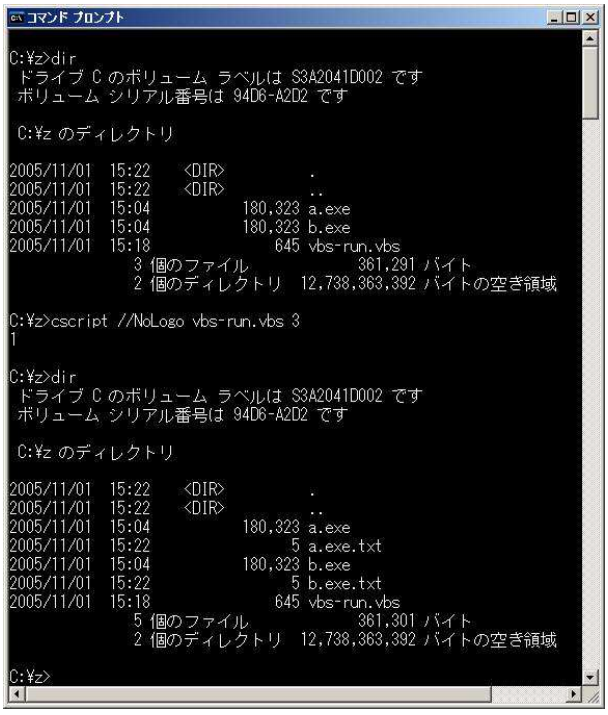

図3．3.1-3：「”cmd.exe /c a.exe” & chrW(124) & ”b.exe**」の結果**

図 3．3.1-2**と同じなので、同じ結果となっている**

(chrW()**関数の確認**)

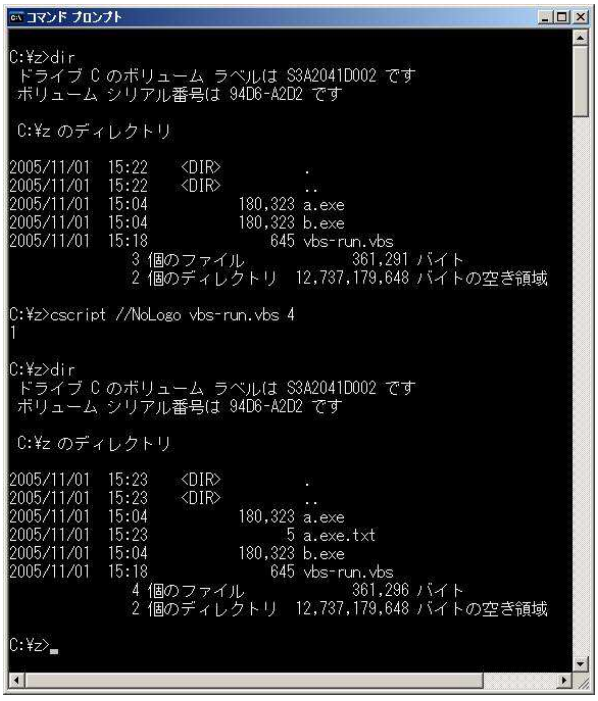

図3．3.1-4：「”cmd.exe /c a.exe” & chrW(166) & ”b.exe”**」の結果**

図 3．3.1-3**とは異なり「|」以下がコマンドとして実行されていない**

3．3.2　WindowsScriptingHost5.6 の WScript.Shell オブジェクトの Exec()メソッド

WSH で外部コマンドを実行する場合、WScript.Shell オブジェクトの Exec メソッドを使うことが多い。Run メソッドは標準出力が取得できないが、Exec メソッドで取得することができる。Exec メソッド自体は、cmd.exe を呼び出していないため「OS コマンドインジェクション」とは無縁の存在ではある(図 3．3.2-1と図 3．3.2-2)。
(引数の強制指定という脅威の可能性は否定しない)

しかし、Exec メソッドから cmd.exe を呼び出して外部コマンドを使えば、「|(0x7c)」(パイプ)などのシェルの機能を利用することができる。

そこで、Exec メソッドは内部的にどのような文字コードを使っているか調査したのが、とである。とで結果が異なっていることから、u00A6 の記号は、シェルのパイプを意味するコード「|(0x7c)」に変換されないことが分かる。
よって、WScript.Shell の Exec メソッドは UNICODE を使ったサニタイジング回避テクニックに対して安全である。

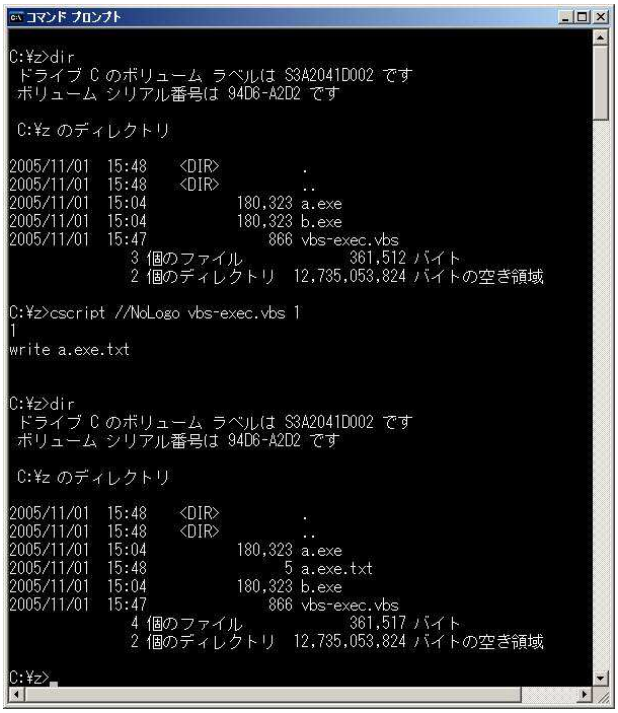

図3．3.2-1：「a.exe|b.exe**」の結果**

b.exe.txt **がないことから「|」以下はコマンドとして実行されていない**

(cmd.exe **を明示的に呼び出す必要がある**)

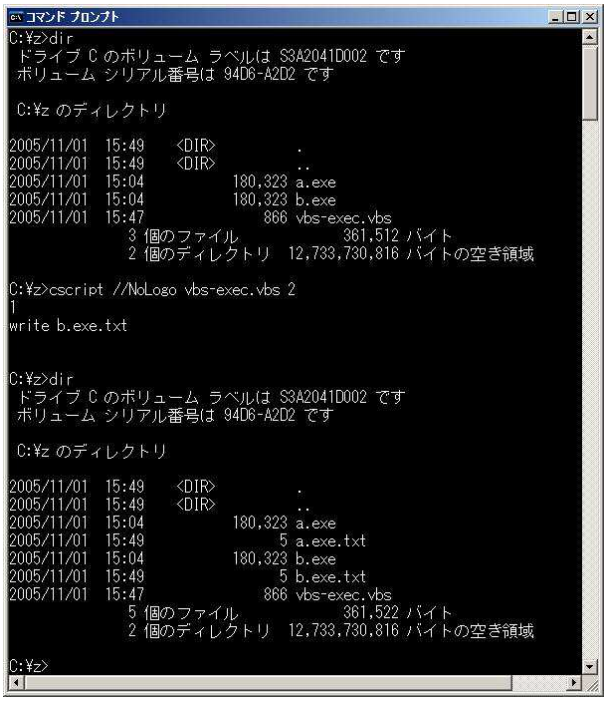

図3．3.2-2：「cmd.exe /c a.exe|b.exe**」の結果**

(cmd.exe **を明示的に呼び出すことで「パイプ」機能を利用できる**)

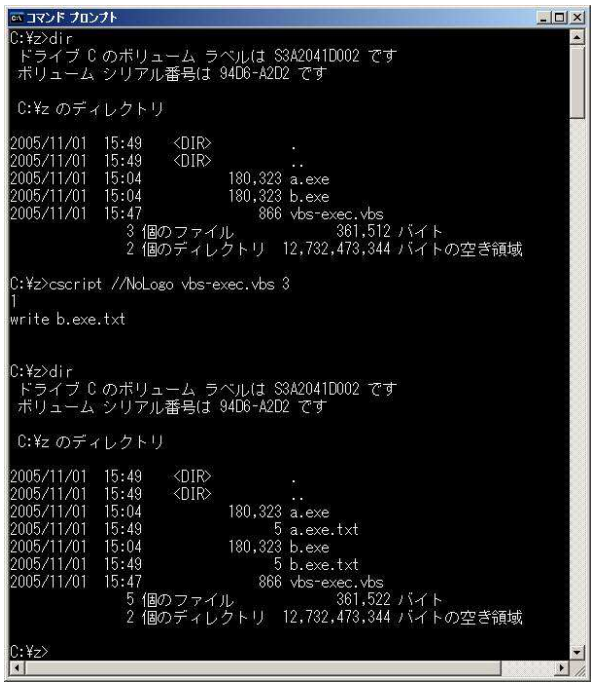

図3．3.2-3：「”cmd.exe /c a.exe” & chrW(124) & ”b.exe**」の結果**

図 3．3.2-2**と同じなので、同じ結果となっている**

(chrW()**関数の確認**)

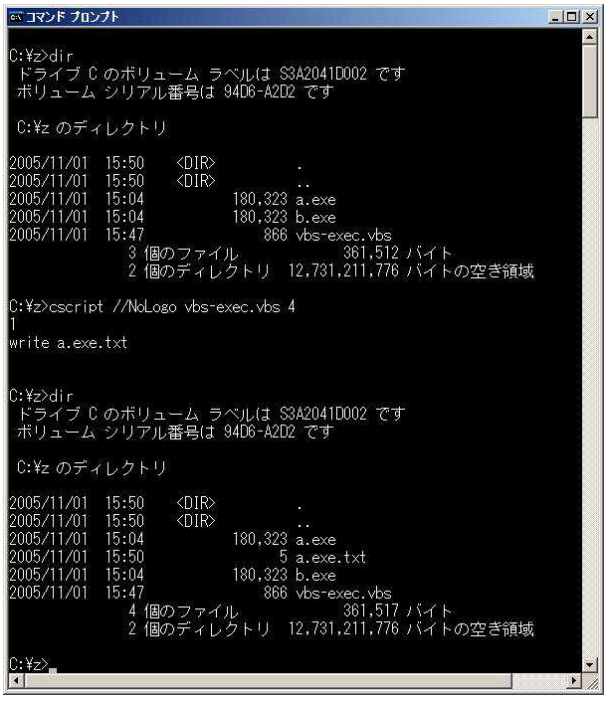

図3．3.2-4：「”cmd.exe /c a.exe” & chrW(166) & ”b.exe”**」の結果**

図 3．3.2-3**とは異なり「|」以下がコマンドとして実行されていない**

3．3.3　VisualBASIC6.0SP6 の Shell()メソッド

VisualBASIC6.0 で外部コマンドを実行する場合、上記の WSH のオブジェクトを利用する場合もあるが、VisualBASIC6.0 には、外部コマンド呼び出しに Shell()メソッドが用意されている。Shell()メソッド自体は、cmd.exe を呼び出していないため「OS コマンドインジェクション」とは無縁の存在ではある(図 3．3.3-1～図 3．3.3-4)。
(引数の強制指定という脅威の可能性は否定しない)

しかし、Shell()メソッドから cmd.exe を呼び出して外部コマンドを使えば、「|(0x7c)」(パイプ)などのシェルの機能を利用することができる。

そこで、Shell()メソッドは内部的にどのような文字コードを使っているか調査したのが、図 3．3.3-5～図 3．3.3-6である。
図 3．3.3-3～図 3．3.3-4と、図 3．3.3-5～図 3．3.3-6では結果が異なっていることから、u00A6 の記号は、シェルのパイプを意味するコード「|(0x7c)」に変換されないことが分かる。
よって、VisualBASIC6.0SP6 の Shell()メソッドは UNICODE を使ったサニタイジング回避テクニックに対して安全である。

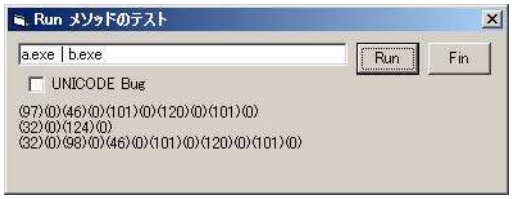

図3．3.3-1：「a.exe | b.exe」を shell に与えた

図3．3.3-2：図 3．3.3-1**の結果**

b.exe.txt **がないことから「|」以下はコマンドとして実行されていない**

(cmd.exe **を明示的に呼び出す必要がある**)

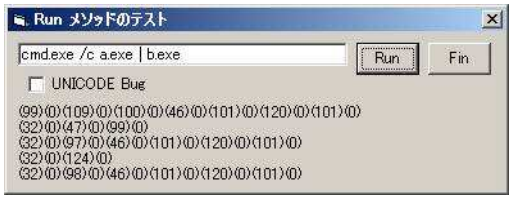

図3．3.3-3：「cmd.exe /c a.exe | b.exe**」を** shell()**に与えた**

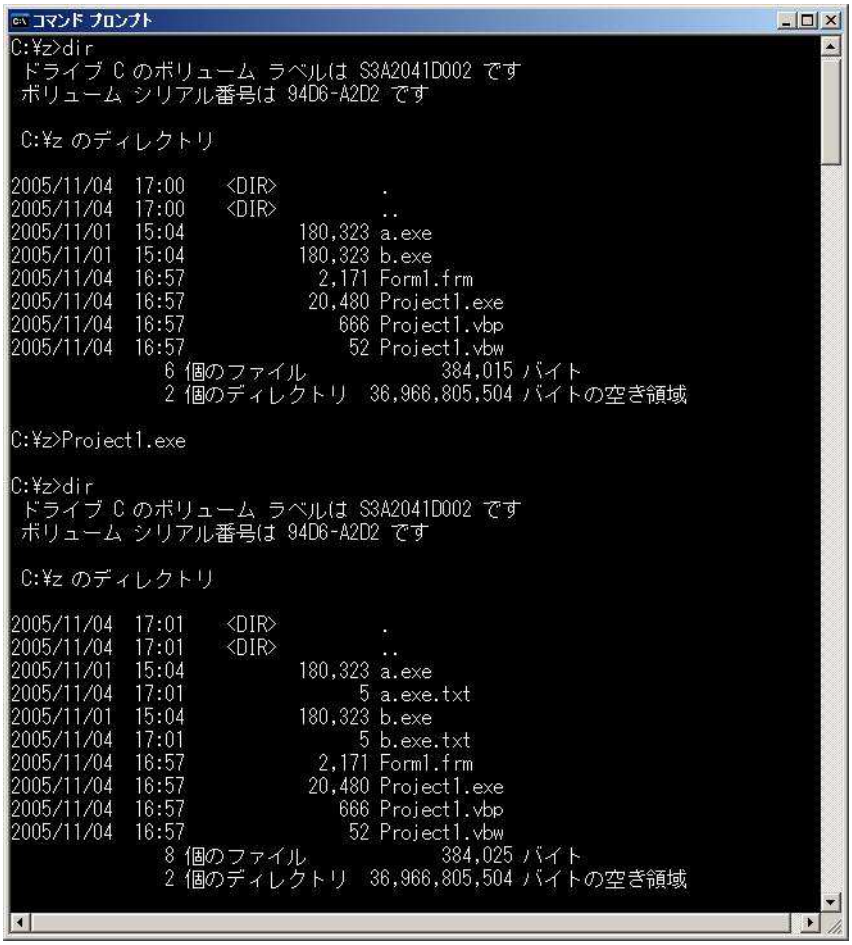

図3．3.3-4：図 3．3.3-3の結果

cmd.exe **を呼び出せば「|」を使うことができる。**

「a.exe.txt**」「**b.exe.txt**」共に作成されたので、「|」以下がコマンドとして実行された。**

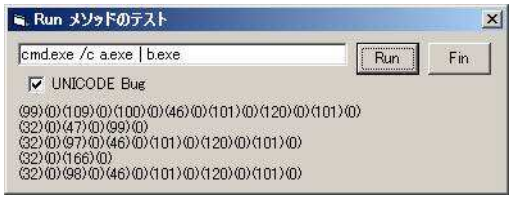

図3．3.3-5：「cmd.exe /c a.exe (u00A6) b.exe**」を** shell()**に与えた**

図3．3.3-6：図 3．3.3-5の結果

「|」を「u00A6」に変換した場合は「a.exe.txt」のみ作成されたので、

UNICODE を使ったサニタイジング回避テクニックは発生しない。

3.4 PostgreSQL/mySQL の SQL 利用時

PostgreSQL/mySQL では、「'」を「¥'」とエスケープすることができる。(「¥」を「¥¥」にエスケープする必要もある)
このような機能があるため「円記号(u00a5)」と「バックスラッシュ(u005c)」による挙動を確認してみた。

3．4.1 Windows 上での PostgreSQL8.0.3(pgODBC7.01.0006)と VisualBASIC6.0SP6

基本的には、デフォルトインストール状態の PostgreSQL に対して、ODBC 経由で VisualBASIC からアクセスした。
VisualBASIC 上では、「'」と「¥」のサニタイジング処理を実施している。

図3．4.1-1：中央上のテキストボックス(「IE」という文字があるボックス)がサニタイジング対象

図3．4.1-2：エラー処理などは実施していないので、「aruyo」「naiyo」ダイアログよりも、エラーとなるかどうかがポイント

図3．4.1-3：「¥」を与える。「UNICODE bug」にチェックを入れると内部的に「バックスラッシュ」を「円記号」に変換する

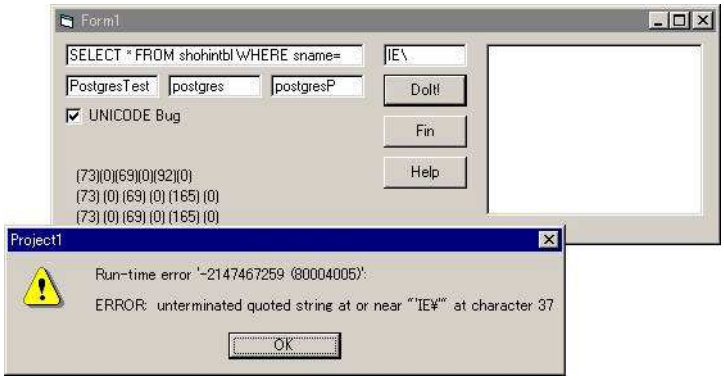

図3．4.1-4：図 3．4.1-3の結果(円記号が「¥(0x5c)」になったため SQL 文法エラーとなる)

図3．4.1-5：「IE¥¥」を与えた(円記号が「¥」になることを見越して「¥¥」としてみた)

図3．4.1-6：図 3．4.1-3の結果

以上より、Windows 版 PostgreSQL の SQL 解釈部は、ANSI 文字で行っているものと推測される。よって、Windows 版 PostgreSQL と VisualBASIC(VBScript も同様だと思われる)の組み合わせでは、UNICODE を使ったサニタイジング回避テクニックにより、SQL インジェクション問題が発現する可能性がある。

3．4.2　Windows 上での mySQL4.1.2alpha(myODBC3.51.07)と VisualBASIC6.0SP6

基本的には、デフォルトインストール状態の mySQL に対して、ODBC 経由で VisualBASIC からアクセスした。
VisualBASIC 上では、「'」と「¥」のサニタイジング処理を実施している。

図3．4.2-1：中央上のテキストボックス(「IE」という文字があるボックス)がサニタイジング対象

図3．4.2-2：エラー処理などは実施していないので、「aruyo」「naiyo」ダイアログよりも、エラーとなるかどうかがポイント

図3．4.2-3：「¥」を与える。「UNICODE bug」にチェックを入れると内部的に「バックスラッシュ」を「円記号」に変換する

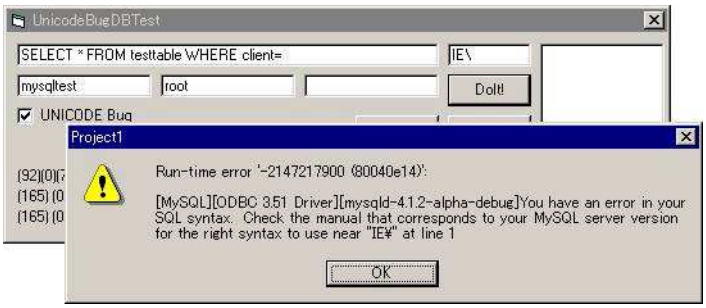

図3．4.2-4：図 3．4.2-3の結果(円記号が「¥(0x5c)」になったため SQL 文法エラーとなる)

図3．4.2-5：「IE¥¥」を与えた(円記号が「¥」になることを見越して「¥¥」としてみた)

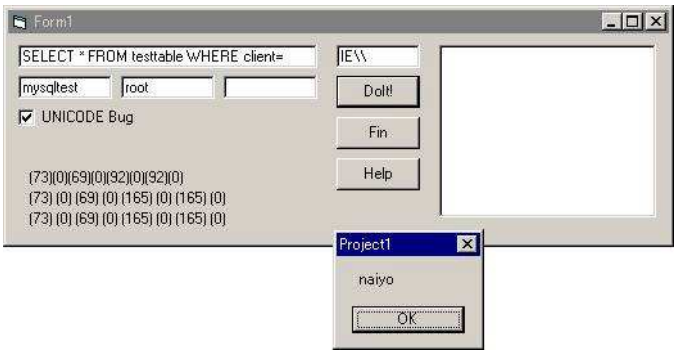

図3．4.2-6：図 3．4.2-5の結果

以上より、Windows 版 mySQL の SQL 解釈部は、ANSI 文字で行っているものと推測される。
よって、Windows 版 mySQL と VisualBASIC(VBScript も同様だと思われる)の組み合わせでは、

UNICODE を使ったサニタイジング回避テクニックにより、SQL インジェクション問題が発現する可能性がある。

3.5 MS-XML COM オブジェクト

XML 文書を DOM オブジェクトとして利用するために、Microsoft は Microsoft XML コアサービスを提供している。
この MS-XML コアサービスでは XPath による XML 文書の検索処理を行うことができる。
XPath による XML 文書の検索では、ユーザから渡される汚染されたデータは、XPath の検索条件部分に配置される場合が一般的であろう。
この MS-XML コアサービスでは、この検索条件部分では、以下のエスケープ処理を行うことが「XPath Injection」対策になる。

- 「¥」→「¥¥」
- 「'」→「¥'」

「¥」が出てきたので、本文書による回避テクニックが使える可能性がある。

3．5.1 Microsoft.XMLDom オブジェクト(VB6SP6) (XML 文書は SJIS)

WSH/ASP/VB などで XML 文書にアクセスする時、使われる DOM オブジェクトである。
ソースコードは、「6.12 MS-XML コアサービスの COM オブジェクトを使った XML 文書検索プログラム (VB6SP6)」を参照。
今回は､(WSH や ASP などと比較して)バイナリ処理が簡単な VisualBASIC6.0SP6 を選択した。

このプログラムでは､「UNICODE Bug」チェックをつけると、「バックスラッシュ(u005c)」を「円記号(u00a5)」に変換する。

最上位のテキストボックスが XML 文書のファイルパスである。
最下位のテキストボックスが XPath による XML 文書の検索結果である。
「Escape」というチェックボックスがオンの場合、XPath の検索条件を「¥」→「¥¥」、「'」「¥'」にエスケープする。

本文書の回避テクニックの本質は、UNICODE→ANSI 変換の縮退と関係があるため、XML 文書は、SJIS とした(図 3．5.1-1)。

検索対象に「'」が含まれている場合、MS-XML コアサービスでは、「'」→「¥'」にエスケープする必要がある(図 3．5.1-2)。
次に検索対象に「¥」が含まれている場合、MS-XML コアサービスでは、「¥」→「¥¥」にエスケープする必要がある(図 3．5.1-3)。
最後に上記の組み合わせであるが、検索対象に「¥'」が含まれている場合、MS-XML コアサービスでは、「¥'」→「¥¥¥'」にエスケープする必要がある(図 3．5.1-4)。
最後に検索文字列中の「バックスラッシュ(0x5c)」を「円記号(u00a5)」に置換してみた結果が、図 3．5.1-5である。
UNICODE を使ったサニタイジング回避テクニックが可能であると推定していたのであるが、図 3．5.1-5を見ても分かるように入力された検索条件中の「u00a5」は「¥¥(0x5c,0x5c) {エスケープされた「¥」}」として内部的に解釈を行っているのではないかと推定される。

結論として、VB/ASP/WSH などから ANSI な XML 文書に対して MS-XML コアサービスを COM を使う場合、UNICODE を使ったサニタイジング回避テクニックに対して安全である。

図3．5.1-1：対象とした XML **文書**(SJIS)

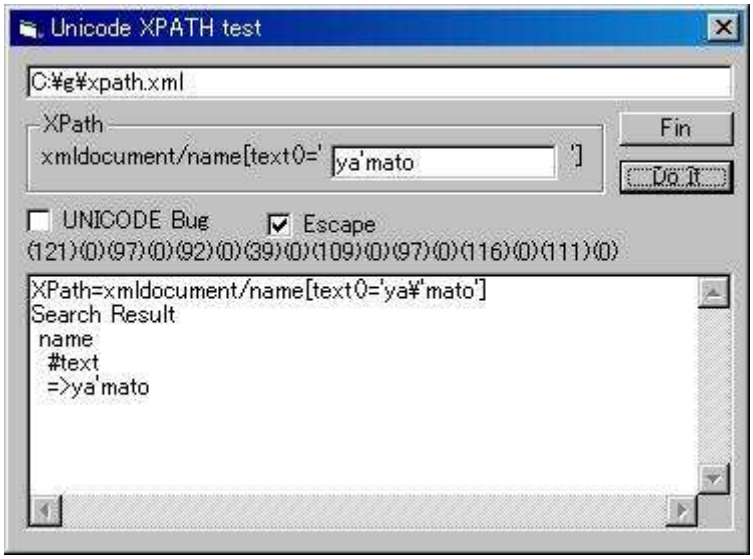

図3．5.1-2：「'」を含んだ検索には「'」を「¥'」にエスケープする

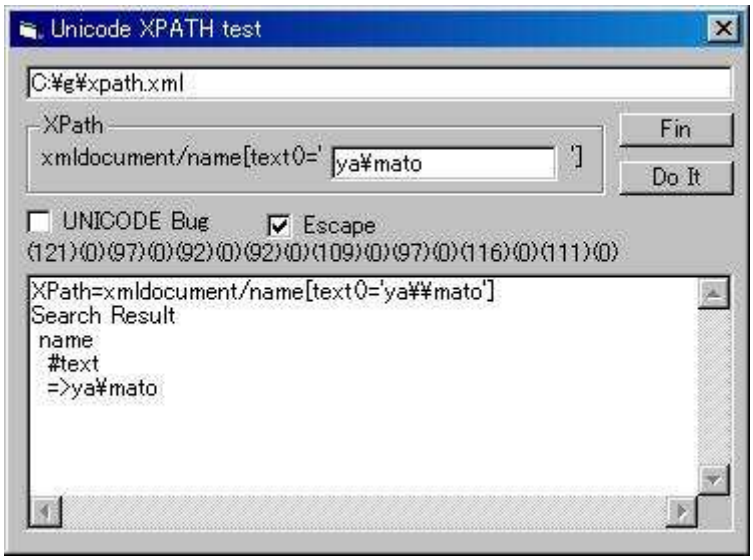

図3．5.1-3：「¥」を含んだ検索には「¥」を「¥¥」にエスケープする

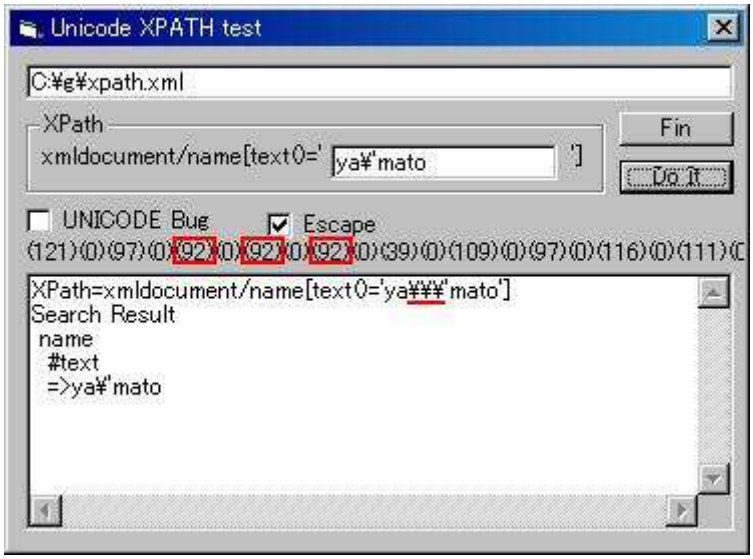

図3．5.1-4：「¥'」を含んだ検索には「¥'」を「¥¥¥'」にエスケープする

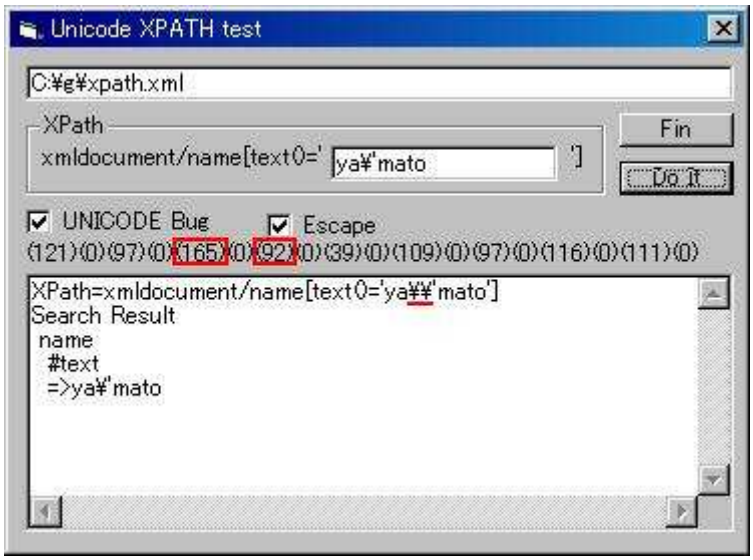

図3．5.1-5：「¥」を「u00a5」にしてみた結果

3.6 LDAP インジェクション

LDAP 検索フィルタに対して、細工したデータを注入することによって、検索条件を改変するセキュテリィ侵害行為は「LDAP インジェクション」と呼ばれている。

RFC2254 に検索フィルタのエスケープ処理が規定されている。
RFC2254 では、「*」「(」「)」「¥」「ヌル文字」を「¥」を前方に付与して 2 文字の 16 進表示にする。と規定されている。
「¥」が登場しているため、UNICODE を使ったサニタイジング回避テクニックが使えそうだが、上記のメタキャラクタを 16 進表示へと変換するため、UNICODE を使ったサニタイジング回避テクニックは利用できない。
基本的には、LDAP インジェクション対策としてのエスケープ処理は、UNICODE を使ったサニタイジング回避テクニックに対して安全であると思われる。

4　アタックベクタ

4.1 Web アプリケーション

以下では、各Webアプリケーションに対してWebブラウザから送り込まれるUNICODEデータはどのように処理されるかを観察する
結論としては、当然のことだが、UNICODE で受け取るような設定(デフォルトの場合や開発者が明示する場合)にしていた場合は、UNICODE を使ったサニタイジング回避テクニックを利用される危険性がでてくる。
老婆心ながら、そもそも Web アプリケーションで使われているモジュール全てが UNICODE 化されていれば本文書の回避テクニックによる脅威はない。

4．1.1 IIS5.1 上の ASP

IIS 上で動く ASP(Active Server Pages)プログラムは、UNICODE をどのように扱っているのかを観察する。
観察した対象は、IIS5.1 (WindowsXP Sp2 日本語版)で観察した。

観察対象は「バックスラッシュ(u005c)」と「円記号(u00a5)」を使い、同一のバイト列に変換されるのか、異なるバイト列として処理されるのかを観察する。

ソースコードは、「6.13 IIS-ASP の文字列データ(String in Variant)のバイト列表示(10進)(ActiveX DLL by VB6SP6)」および「6.14 IIS-ASP で Web ブラウザから受け取ったリクエストのバイト列表示(IIS-ASP + VBScript)」である。

通常の ASP 開発では、特に「CodePage」の設定は行わないだろう。
test.asp は「CodePage」を設定していない ASP プログラムである。この test.asp に対して、「%u005c%u00a5」を与えた結果が図 4．1.1-1である。
UNICODE の「円記号(u00a5)」が、ASP プログラム中で既に「バックスラッシュ(u005c)」に縮退しているため、本文書の回避テクニックは利用できない。
つぎに「%a5」を与えた結果が図 4．1.1-2である。文字化けしているため、本文書の回避テクニックは利用できない。最後に UTF-8 表現である「%2c%a5」を与えた結果が図 4．1.1-3である。バイナリ的には「円記号(u00a5)」ではないため、本文書のテクニックは利用できない、と判断できるが、画面上には「円記号」が表示されている。もう少し、検討の余地がある。

以上の結果から、ASP 開発者が特別に「CodePage」を設定しない場合、Web ブラウザから与えられたデータは、非 UNICODE 文字として正規化されると評価しても問題ないだろう(UTF-8 書式での入力にはまだ検討の余地がある)。
つまり、このような ASP プログラムをインターフェイスとして本文書の UNICODE を使ったサニタイジング回避テクニックは使用できない、ということである。

ASP プログラムで UNICODE を扱う場合、「CodePage」を明示的に指定する方法がある。「testu.asp」では「UTF-8」となるように明示している。
testu.asp に対して、「%u00a5」「%a5」「%c2%a5」を与えた結果が図 4．1.1-4～図 4．1.1-6である。
図のように、「バックスラッシュ」と「円記号」が異なるバイト列で内部的に扱われているのが分かる。このように、ASP プログラムで明示的に「CodePage」を「UTF-8」を指定している場合、この ASP プログラムを経由した攻撃で UNICODE を使ったサニタイジング回避テクニックが利用できる、ということになる。

図4．1.1-1：「codepage」はデフォルトのままで「%u00a5」を与えた場合の結果

図4．1.1-2：「codepage」はデフォルトのままで「%a5」を与えた場合の結果

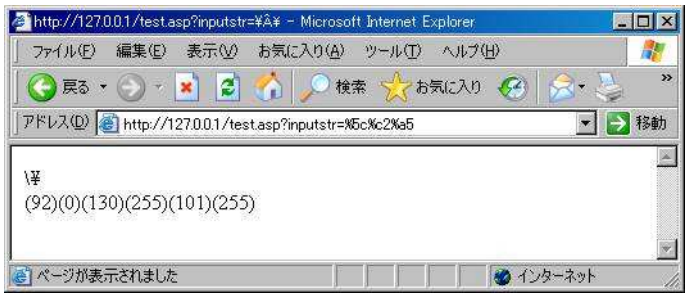

図4．1.1-3：「codepage」はデフォルトのままで「%c2%a5」を与えた場合の結果

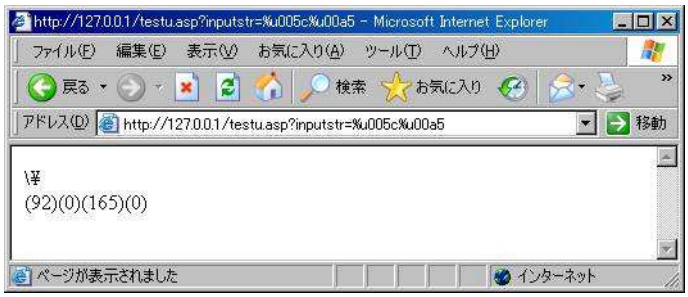

図4．1.1-4：ASP 上で明示的に「codepage」を UTF-8 に設定し、「%u00a5」を与えた場合の結果

図4．1.1-5：ASP 上で明示的に「codepage」を UTF-8 に設定し、「%a5」を与えた場合の結果

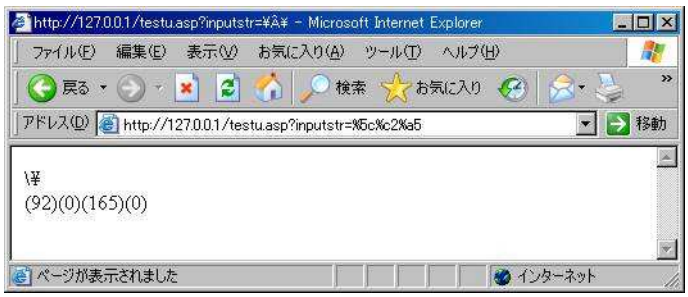

図4．1.1-6：ASP 上で明示的に「codepage」を UTF-8 に設定し、「%2c%a5」を与えた場合の結果

(※) 本文書とは無関係であるが、ASP でコードページを指定する場合の MS-KB を見つけたのでリンクする

- コードページが UTF8 に設定されていると ASP スクリプトでタイプライブラリを使用できない

  http://support.microsoft.com/default.aspx?scid=kb%3Bja%3B294833

4．1.2　ASP.NET

IIS 上で動く ASP.NET プログラムは、UNICODE をどのように扱っているのかを観察する。
観察した対象は、IIS5.0 + .NET Framework 1.1 + (C# または VB.NET) + WebMatrix 0.6 (Windows2000 Sp4 日本語版)で観察した。

観察対象は「バックスラッシュ(u005c)」と「円記号(u00a5)」を使い、同一のバイト列に変換されるのか、異なるバイト列として処理されるのかを観察する。

ソースコードは、「6.15 ASP.Net で Web ブラウザから受け取ったリクエストのバイト列表示(C#)」および「6.16 ASP.Net で Web ブラウザから受け取ったリクエストのバイト列表示(VB.NET)」である。

通常の ASP.NET 開発でも特に「CodePage」の設定は行わないだろう。HexDispCS.aspx/HexDispVB.aspx でも特にコードページの指定はしていない。
また、テキストボックスへ与えたデータは POST されるため、POST データ改変のために sPortRedirector2 を使用した。
sPortRedirector2 を使用し、テキストボックスの内容に「%u00a5」「%a5」「%c2%a5」を付与し ASP.NET に与えた結果が図 4．1.2-1～図 4．1.2-12である。
図(図 4．1.2-2、図 4．1.2-6、図 4．1.2-8、図 4．1.2-12)から読み取れる通り、「%u00a5」「%c2%a5」を与えた時に「バックスラッシュ」と「円記号」が異なるバイト列として認識している。
このように特にコードページを指定していない場合、UNICODE を使ったサニタイジング回避テクニックが利用できる、ということになる。

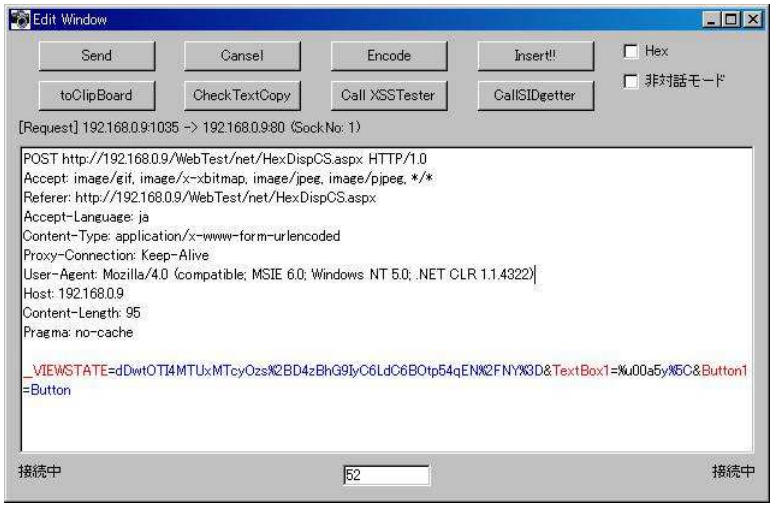

図4．1.2-1：POST されるデータを改変しテキストデータの先頭に「%u00a5」を付与してサーバへ送る(C#)

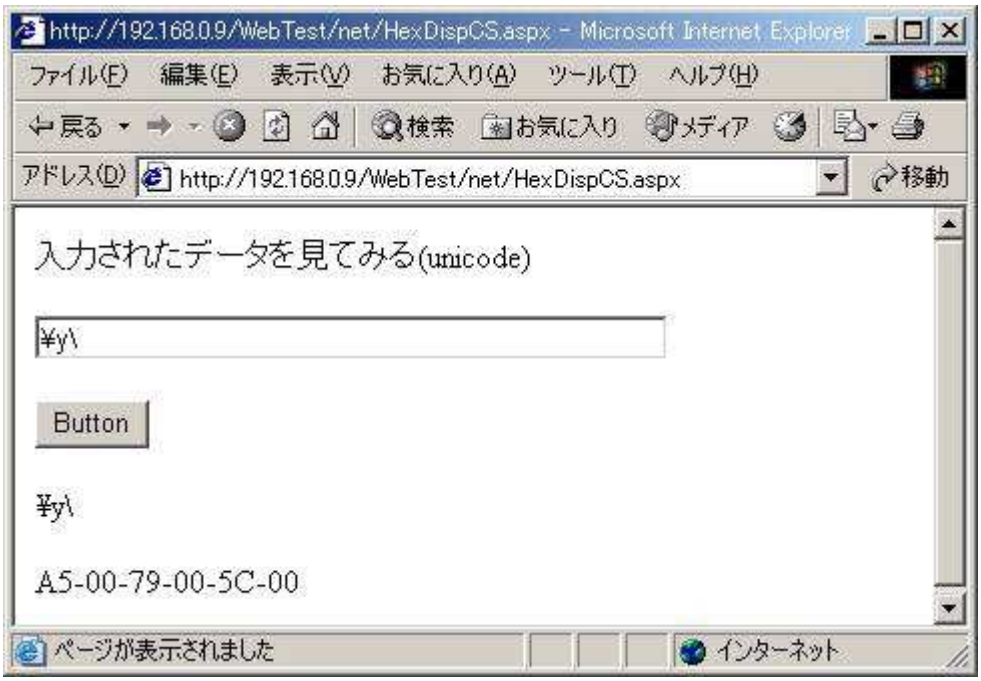

図4．1.2-2：図 4．1.2-1の結果(C#)

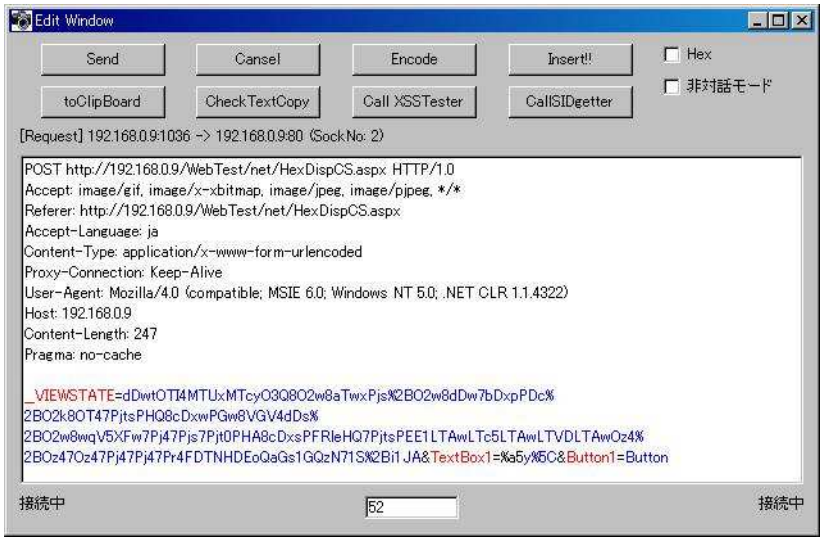

図4．1.2-3：POST **されるデータを改変しテキストデータの先頭に「**%a5**」を付与してサーバへ送る**(C#)

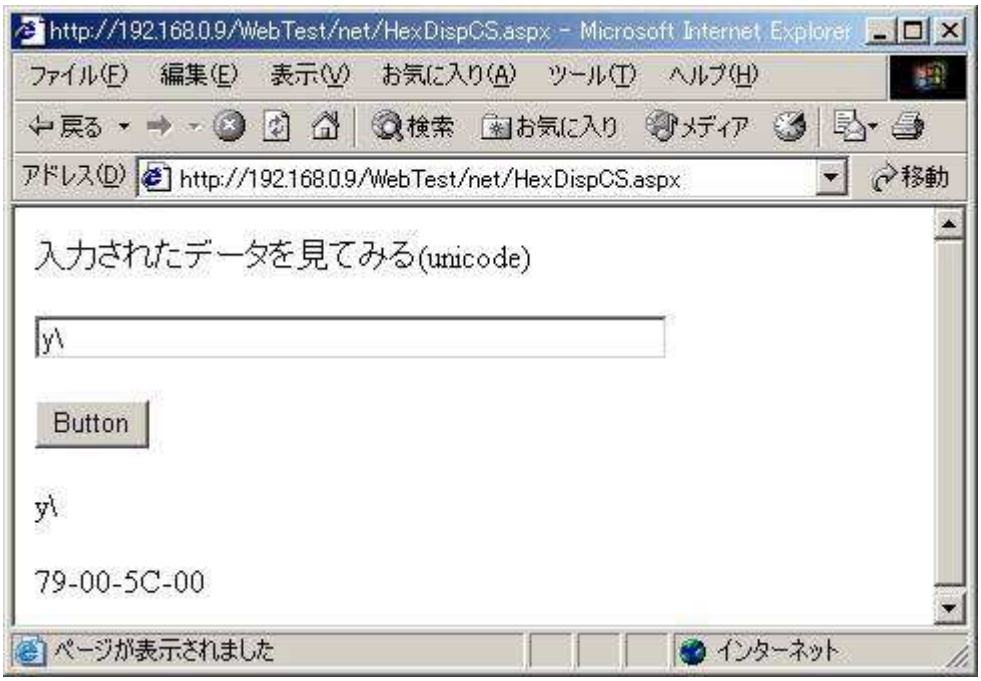

図4．1.2-4：図 4．1.2-3の結果(C#)

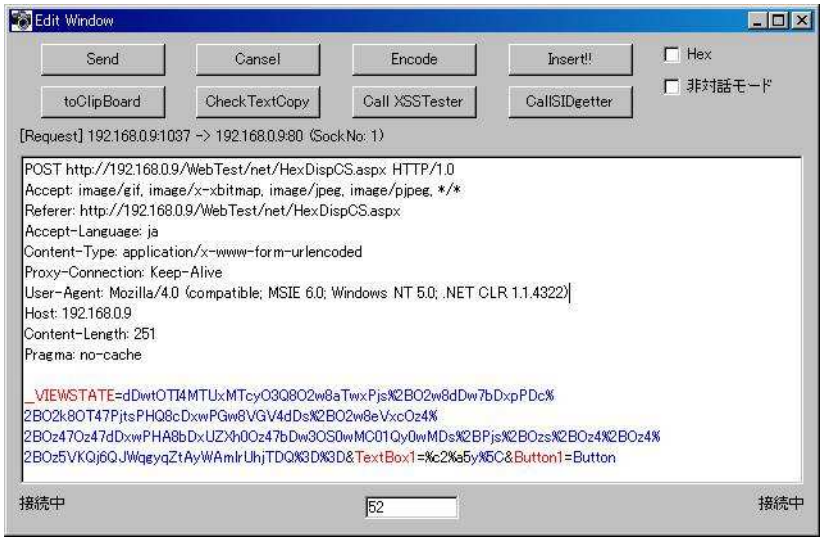

図4．1.2-5：POST **されるデータを改変しテキストデータの先頭に**

0xa5 **の** UTF-8 **表現である「**%c2%a5**」を付与してサーバへ送る**(C#)

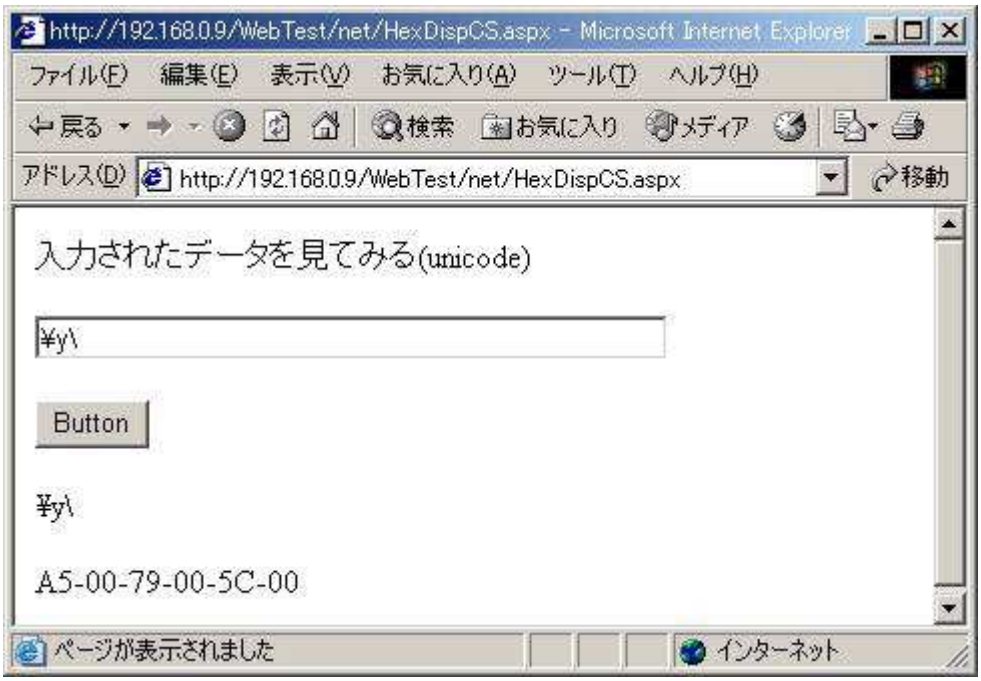

図4．1.2-6：図 4．1.2-5の結果(C#)

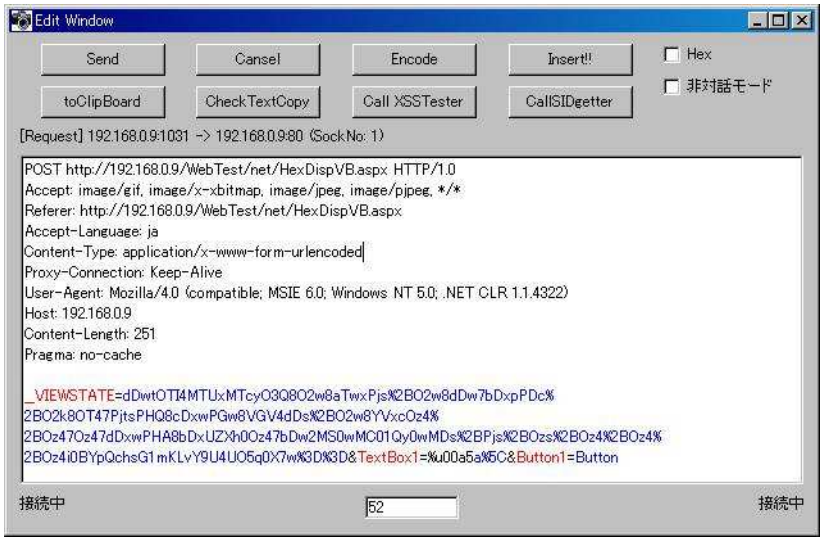

図4．1.2-7：POST **されるデータを改変しテキストデータの先頭に「**%u00a5**」を付与してサーバへ送る**(VB.NET)

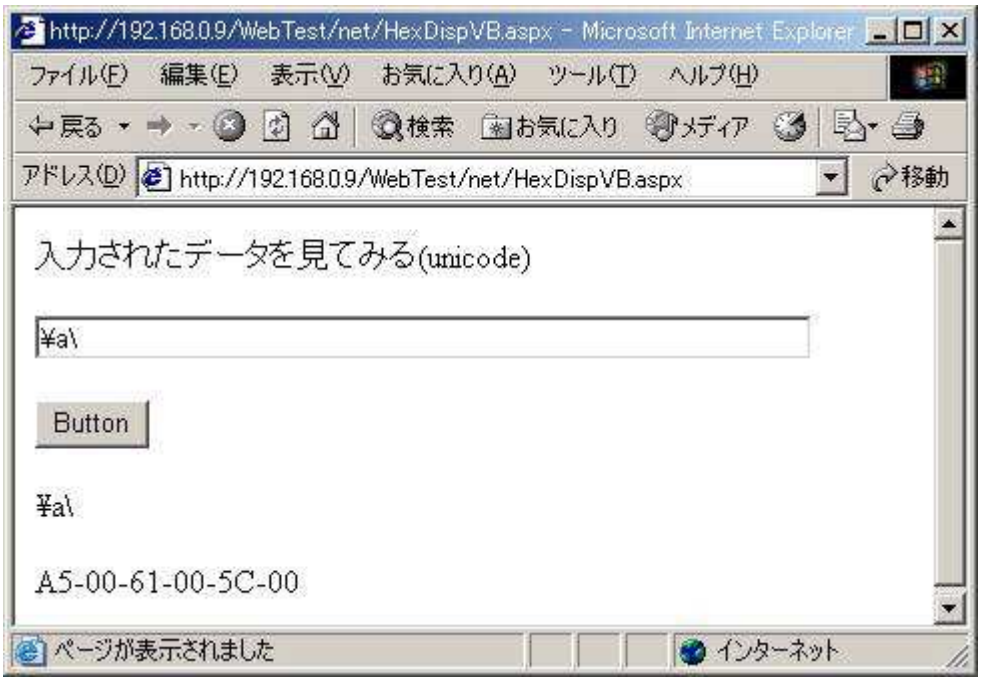

図4．1.2-8：図 4．1.2-7の結果(VB.NET)

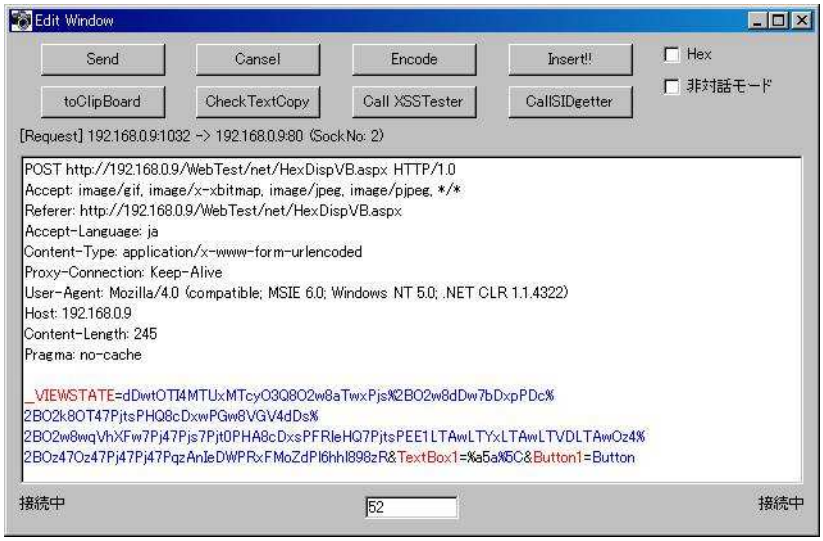

図4．1.2-9：POST **されるデータを改変しテキストデータの先頭に「**%a5**」を付与してサーバへ送る**(VB.NET)

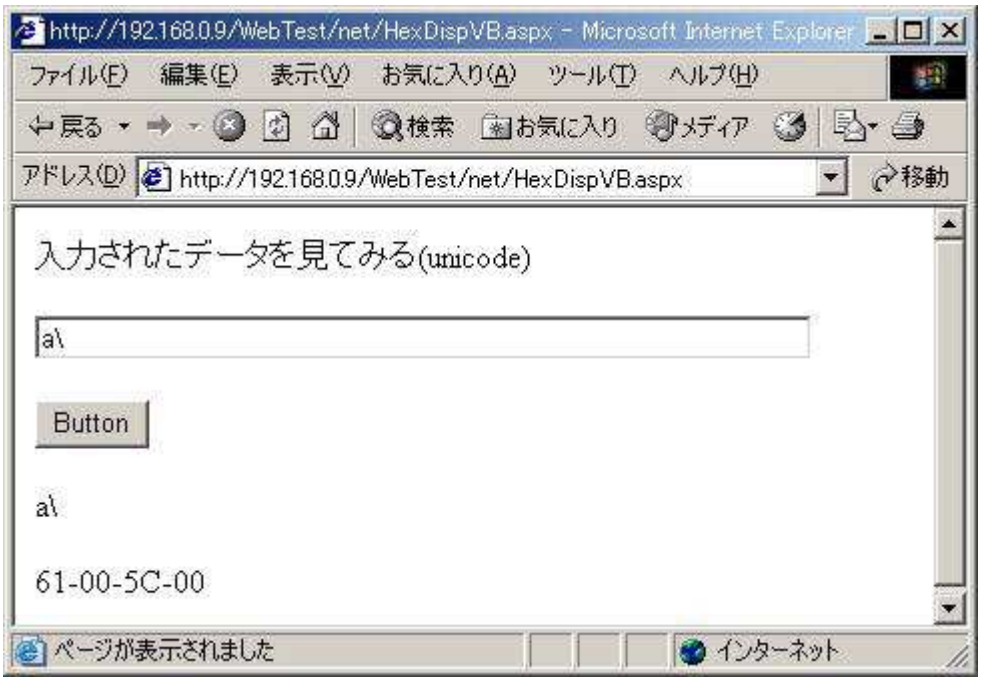

図4．1.2-10：図 4．1.2-9の結果(VB.NET)

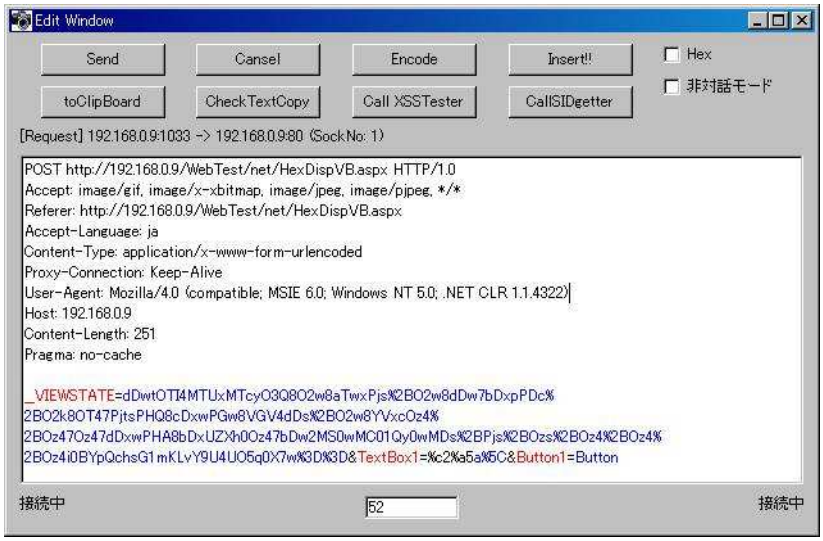

図4．1.2-11：POST **されるデータを改変しテキストデータの先頭に**

0xa5 **の** UTF-8 **表現である「**%c2%a5**」を付与してサーバへ送る**(VB.NET)

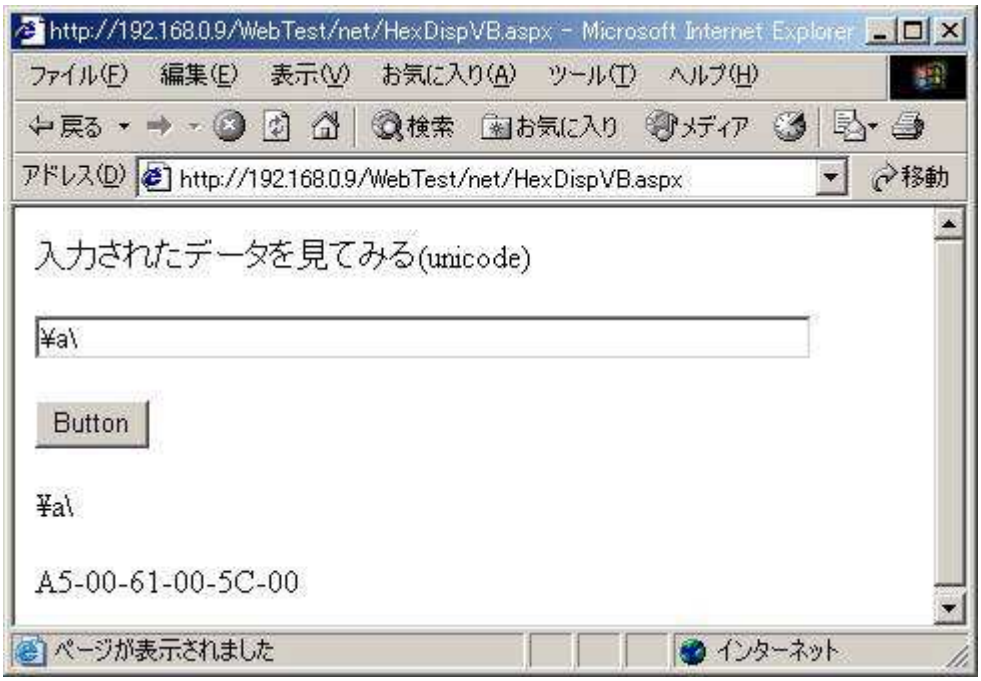

図4．1.2-12：図 4．1.2-11の結果(VB.NET)

4．1.3　IIS + CGI

IIS 上で動く(「_UNICODE」または「_MBCS」**オプションでコンパイルされた)**CGI **プログラムに対して** UNICODE **を送り込んだ場合、**IIS **は** CGI **に対してどのようにデータを受け渡すかを観察した。**
IIS5.1 + (WindowsXP SP2 **日本語版)である。**

送り込む HTTP **リクエスト作成のためにバイナリエディタ**「Stirling ver1.31」**を使用し、通信プログラムとして**「Netcat1.10[NT]」**を使用した。**

CGI **のソースコードは、**「6.17 CGI **として受け取ったデータのバイト列表示プログラム」である。これを**「_UNICODE」**オプションでコンパイルした**「testCgiU.exe」。「_MBCS」**オプションでコンパイルした**「testCgiA.exe」**を用いた。**

ここでの観察目的は、IIS **が** CGI **プログラムの属性(**_UNICODE **または**_MBCS **のどちらでコンパイルされたものであるか)を検出して、相応のエンコード処理を行って** CGI **プログラムへクエリー文字列やコマンドライン引数、フォームデータを渡すかどうかというのが観察目的である。**

UNICODE プログラム「testCgiU.exe」に対しての実験結果が図 4．1.3-1～図 4．1.3-8である。

図 4．1.3-1を見ると「%u00a5」という「円記号(u00a5)」が、コマンドライン引数では ANSI 空間の「バックスラッシュ(u005c)」に縮退している点である。この方法では、本文書のサニタイジング回避テクニックは使用できない。
図 4．1.3-3では、標準入力(ポストされるフォームデータ)に対してのみ「円記号(u00a5)」として扱われている。この方法の場合、フォームデータについてのみ、本文書のサニタイジング回避テクニックが使用可能である。
図 4．1.3-3や図 4．1.3-5でのクエリー文字列やコマンドライン引数での文字化けが少し気がかりであるが、本文書の主旨ではないため割愛する。

図 4．1.3-9～図 4．1.3-13は、ANSI プログラム「testCgiA.exe」に対して行った実験結果である。当然であるが、ANSI プログラムであるためアタックベクタとしては使用できない。
しかし、ANSI な CGI プログラムの前で UNICODE な ISAPI フィルタでサニタイジングをしている、というような想定をした場合、「UNICODE 上でサニタイジング→ANSI プログラム」となるので、本文書の回避テクニックが利用可能である可能性がある。ISAPI フィルタと CGI プログラムとの関係については本項目のテーマではない。

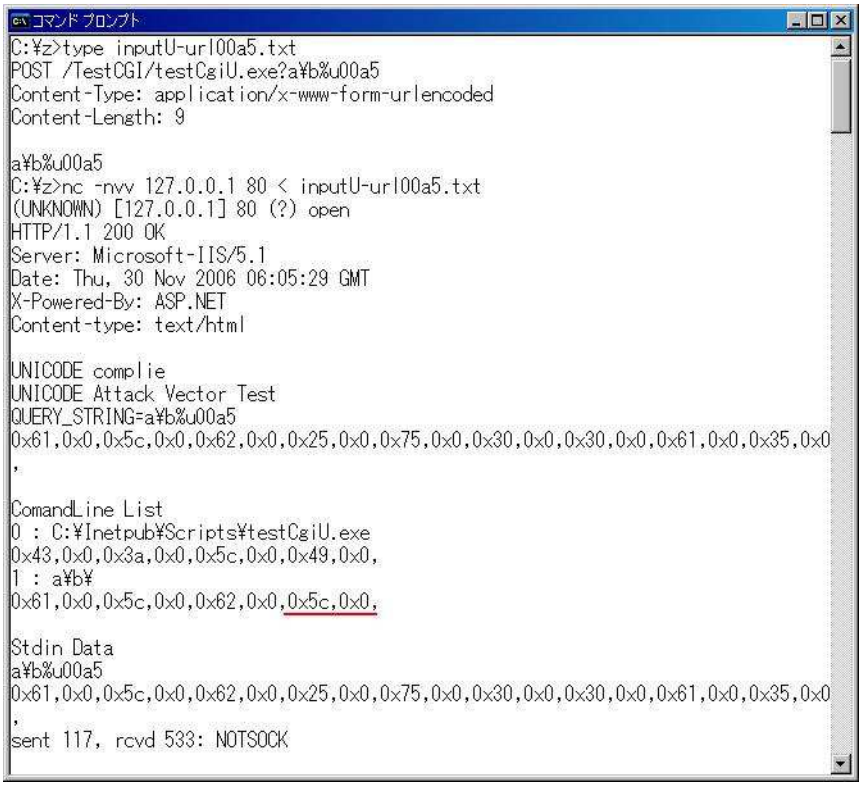

図4．1.3-1：testCgiU.exe に対して「%u00a5」を送った結果。

CGI のコマンドライン引数では ANSI 空間へ縮退して渡されている。

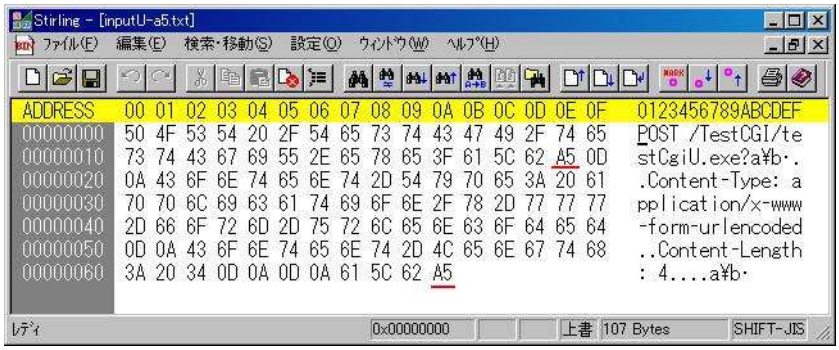

図4．1.3-2：testCgiU.exe **に対して「**0xa5**」を送るためのリクエスト**

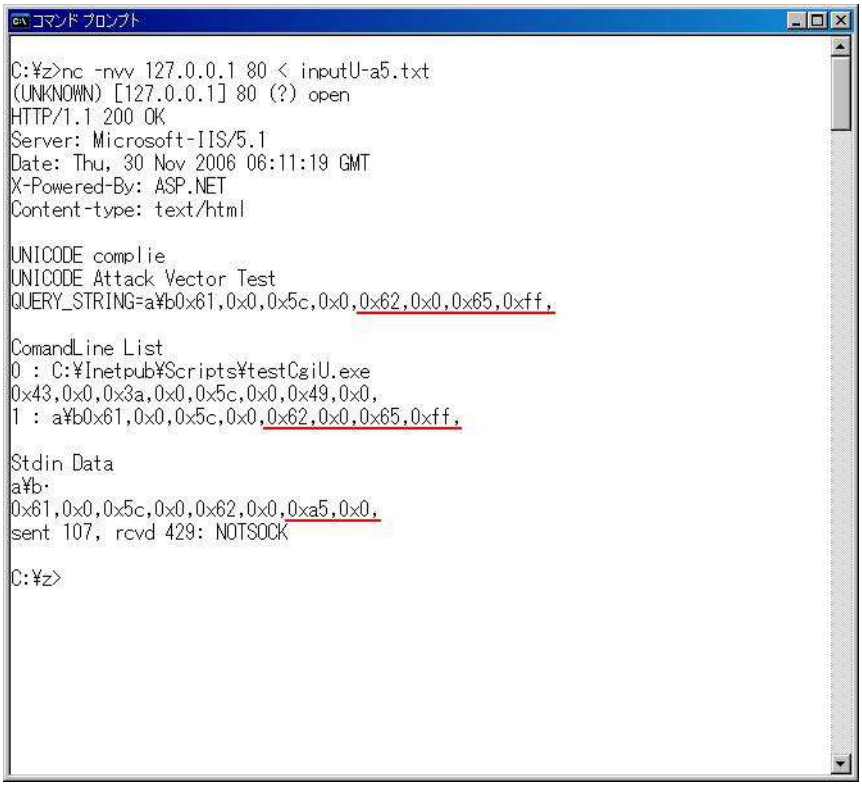

図4．1.3-3：図 4．1.3-2**の結果。**

クエリー文字列とコマンドライン引数では文字化けを起こしているが、標準入力では UNICODE **の円記号として扱われている。**

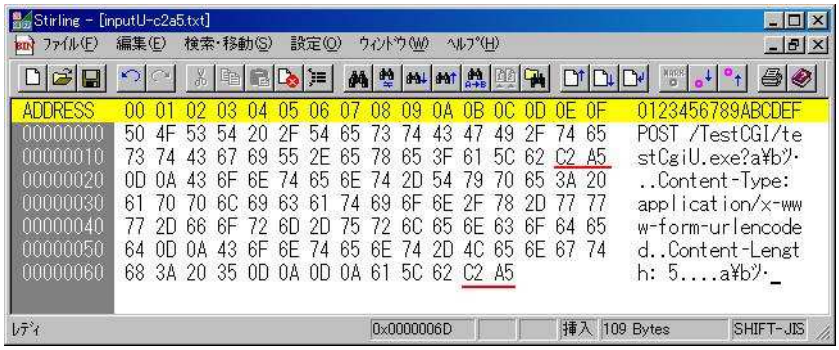

図4．1.3-4 : testCgiU.exe **に対して「**0xc2,0xa5**」を送るためのリクエスト**

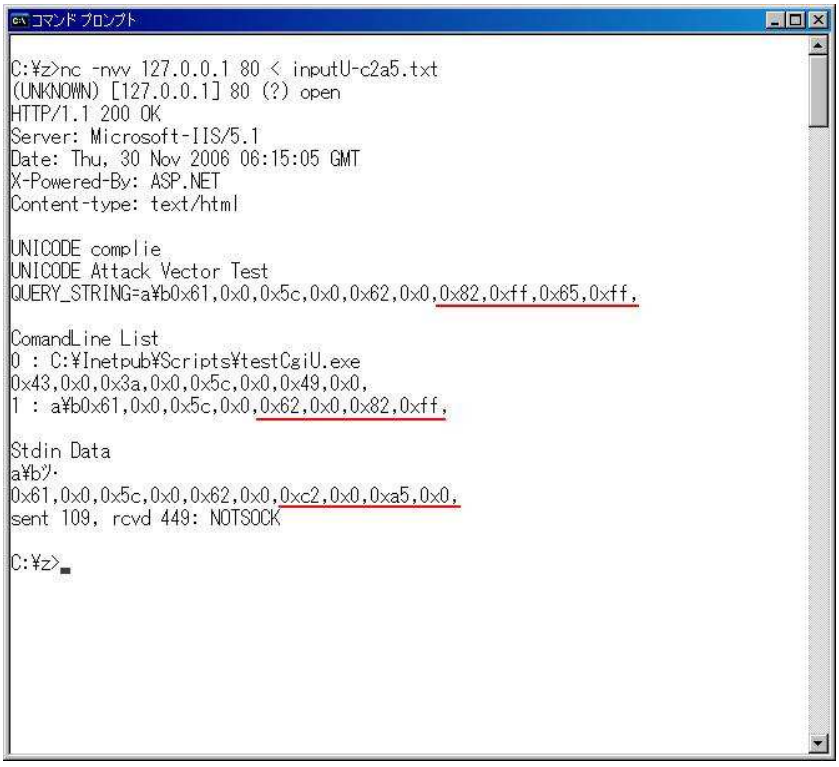

図4．1.3-5 : 図 4．1.3-4**の結果。**

UNICODE **の円記号としてエンコードされた箇所がない**

図4．1.3-6 : この図のように UTF-16 **を直送してみる**

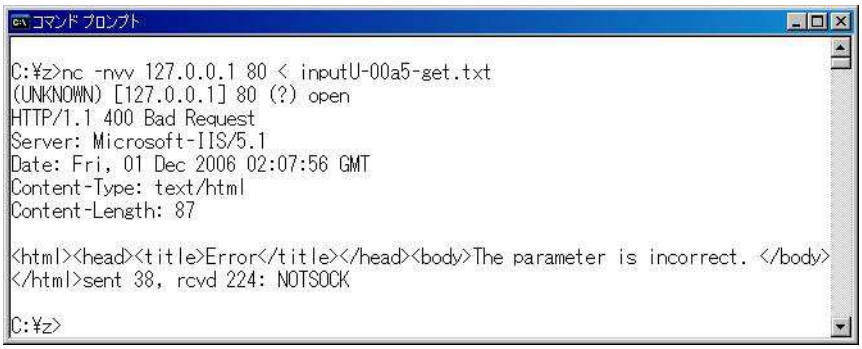

図4．1.3-7：図 4．1.3-6の結果。エラーである。どうも IIS は「0x00」を拒絶するようだ。

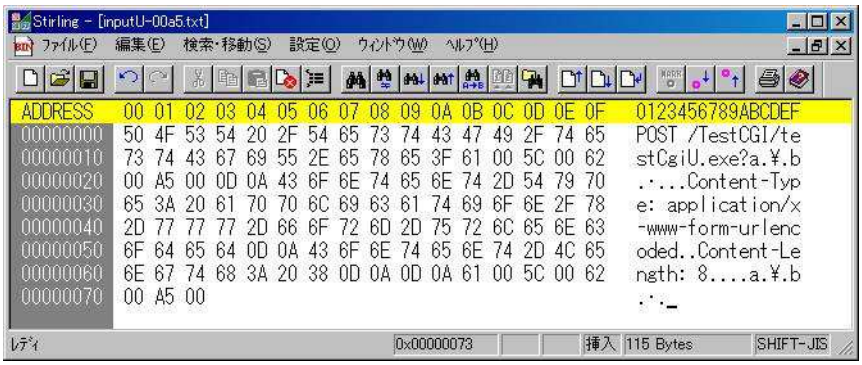

図4．1.3-8：この図のように UTF-16 を直送すると無反応になった(Post データ待ちの状態であると推定される)

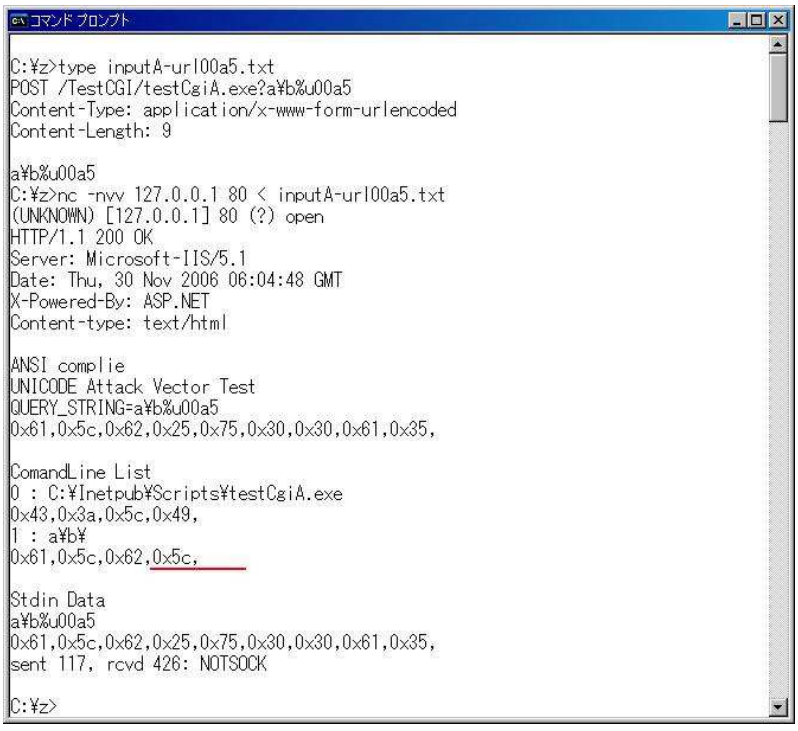

図4．1.3-9：testCgiA.exe に対して「%u00a5」を送った結果

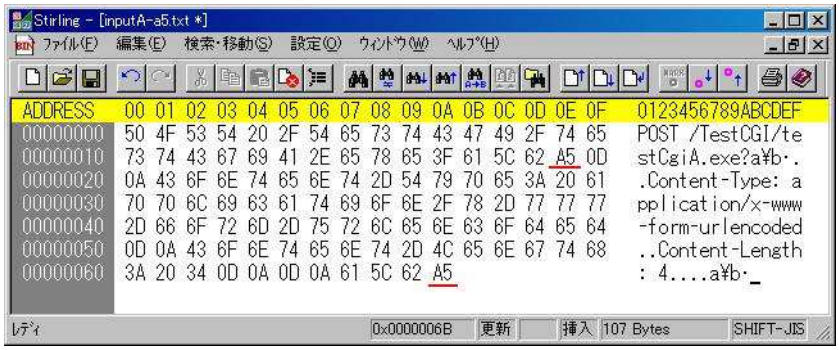

図4．1.3-10 : testCgiA.exe **に対して「**0xa5**」を送るためのリクエスト**

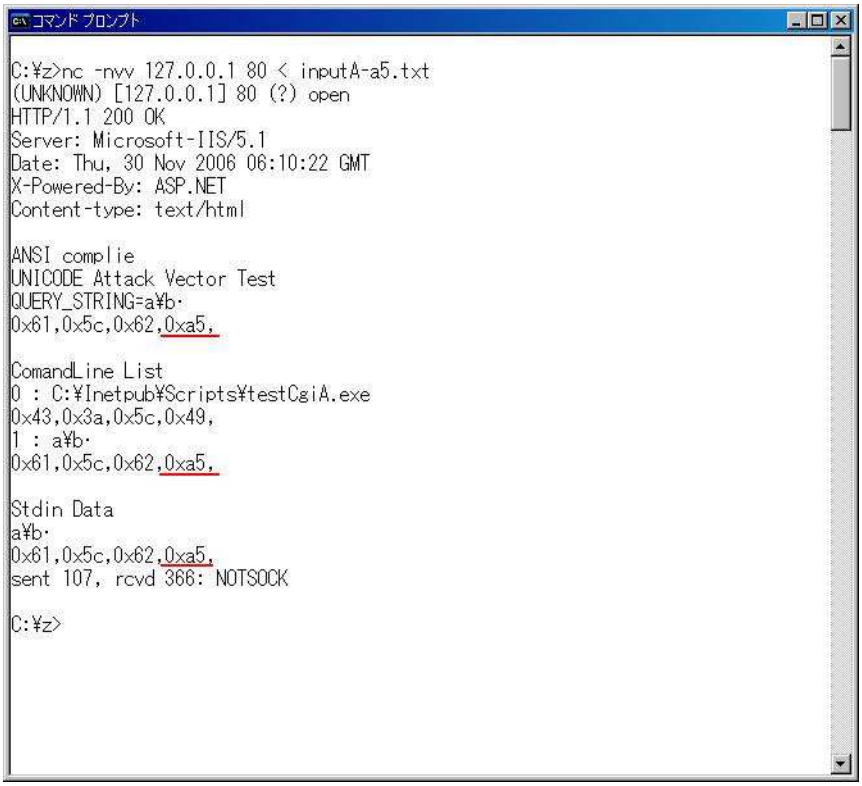

図4．1.3-11：図 4．1.3-2**の結果**

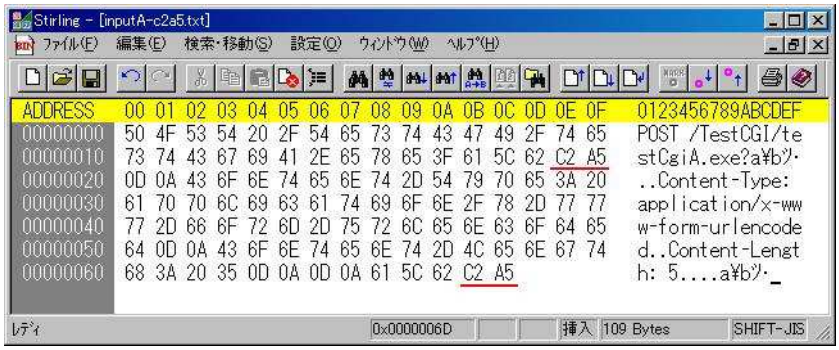

図4．1.3-12 : testCgiA.exe **に対して「**0xc2,0xa5**」を送るためのリクエスト**

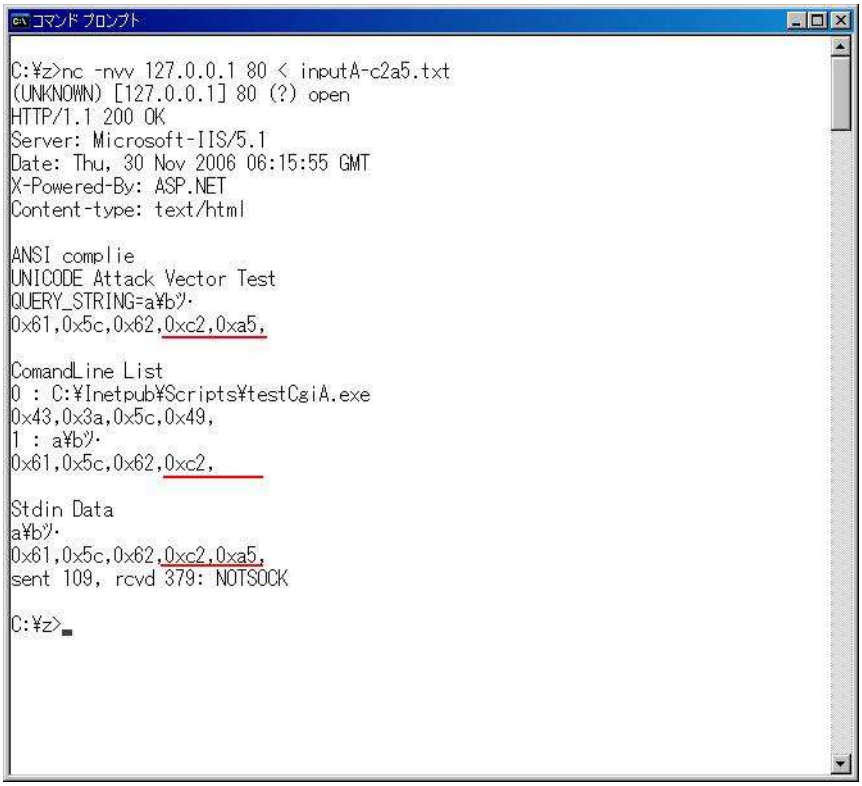

図4．1.3-13 : 図 4．1.3-4**の結果**

4．1.4　JavaServlet

Java **での** Web **アプリケーションの場合について観察した。**
観察した Java **環境は、**Windows2000SP4 **日本語版で動作する** JDK1.5.08 + Tomcat5.5.17 **である。ちなみに** Eclipse3.1.2 **で開発した。**

Java の場合、Request オブジェクトに対して setCharacterEncoding()メソッドによって文字コードを指定することができる。

行ったテスト内容は、Post されるデータまたはクエリ文字列に「円記号」を示す「%u00a5」、「%a5」及び「0xa5」の UTF-8 表現である「%c2%a5」を与えた。
Post データの改変には sPortRedirector2 を使用した。

- UTF-8 を使用した場合の図が、図 4．1.4-1～図 4．1.4-6である。
  当然だか、「バックスラッシュ」と「円記号」は異なるバイト列で保持されている。
  よって、UNICODE を使ったサニタイジング回避テクニックが利用できる、ということになる。
- 文字コードに「Windows-31J」を指定した場合が、図 4．1.4-7～図 4．1.4-12である。
  図を見てみると文字化けはあるが(後日に別の問題として評価してみたいがそれは別の話)、入力データとして与えた「円記号」がそのまま評価されていないことから、UNICODE を使ったサニタイジング回避テクニックを利用できない、ということになる。

Java では、ASP/ASP.NET で通用した「%uxxxx」という表現が通用していないことに注意する必要がある。
また、Post する UNICODE 文字は「%a5」ではなく、UTF-8 表現の「%c2%a5」でなければならないようだ。

今回の観察には Tomcat5 系を使用したが、Tomcat 4 系からの仕様変更がある。それは Tomcat5 系では、クエリー文字列のデフォルト文字コードは「UTF-8」となり、setCharacterEncoding() を適用しない。ということである。
そのことのテストをしてみたのが、図 4．1.4-13～図 4．1.4-15である。
図 4．1.4-13～図 4．1.4-15では、doPost()メソッドではなく、doGet()メソッドを使ってクエリー文字列を request オブジェクトとして取得した。また setCharacterEncoding()メソッドによって文字コードを「Windows-31J」にしている。
結果的に setCharacterEncoding()で「Windows-31J」としていてもクエリー文字列からの入力データを使って、UNICODE によるサニタイジング回避テクニックが利用できる、ということを図 4．1.4-14は示している。
ここで、想定外の現象としてクエリー文字列に対しては「u00a5」は UTF-8 表現ではなくそのまま「%a5」だけでよいという点であるが、これは本文書の回避テクニックの内容とは異なるので、これ以上は解析しない。

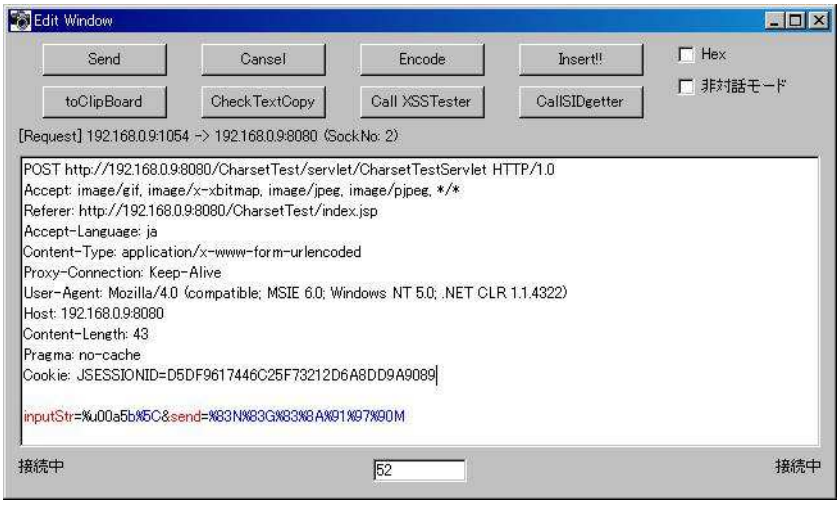

図4．1.4-1 : POST **されるデータを改変しテキストデータの先頭に「**%u00a5**」を付与してサーバへ送る**

図4．1.4-2 : 図 4．1.4-1**の結果**(Java_UTF8)

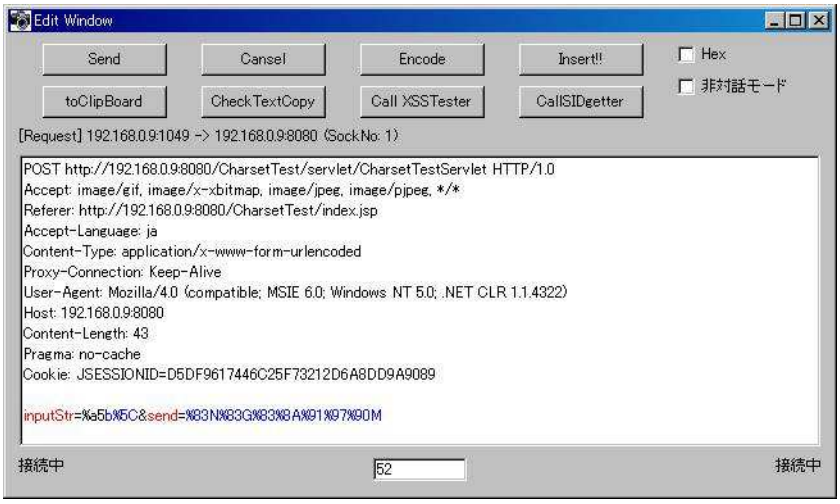

図4．1.4-3 : POST **されるデータを改変しテキストデータの先頭に「**%a5**」を付与してサーバへ送る**

図4．1.4-4：図 4．1.4-3の結果(Java_UTF8)

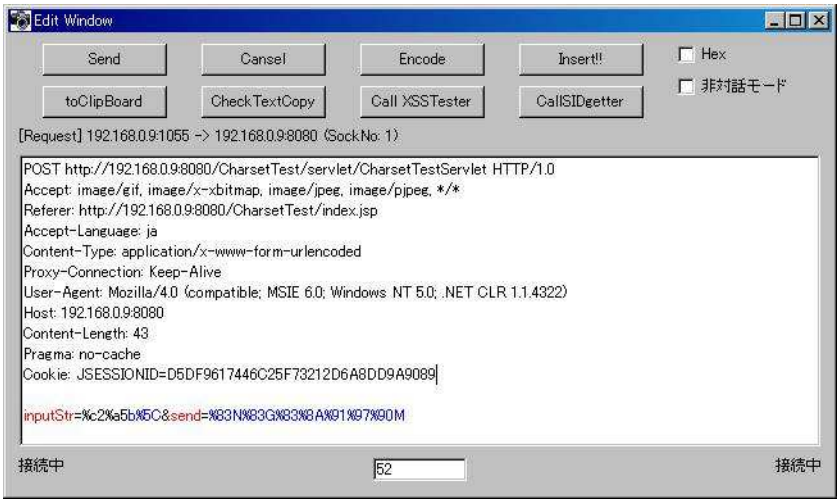

図4．1.4-5：POST **されるデータを改変しテキストデータの先頭に**

0xa5 **の** UTF-8 **表現である「**%c2%a5**」を付与してサーバへ送る**

図4．1.4-6：図 4．1.4-5の結果(Java_UTF8)

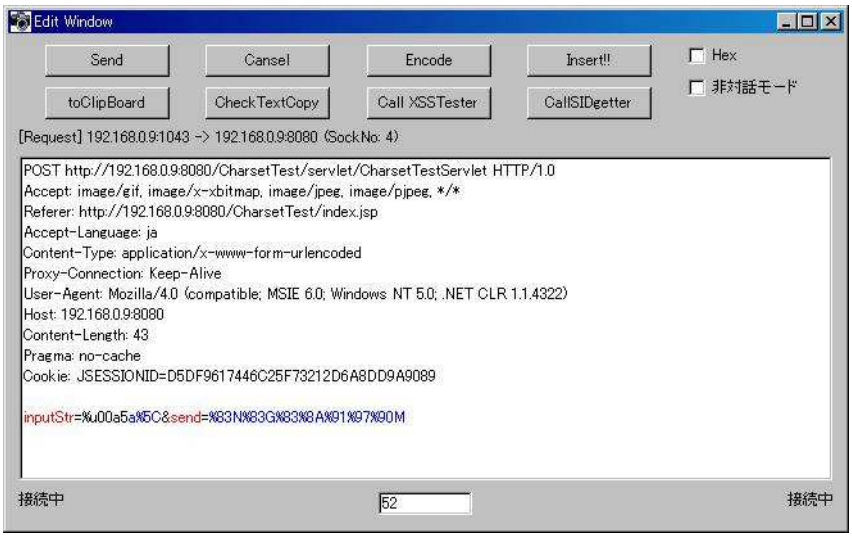

図4．1.4-7 : POST **されるデータを改変しテキストデータの先頭に「**%u00a5**」を付与してサーバへ送る**

図4．1.4-8 : 図 4．1.4-7**の結果**(Java_Windows-31J)

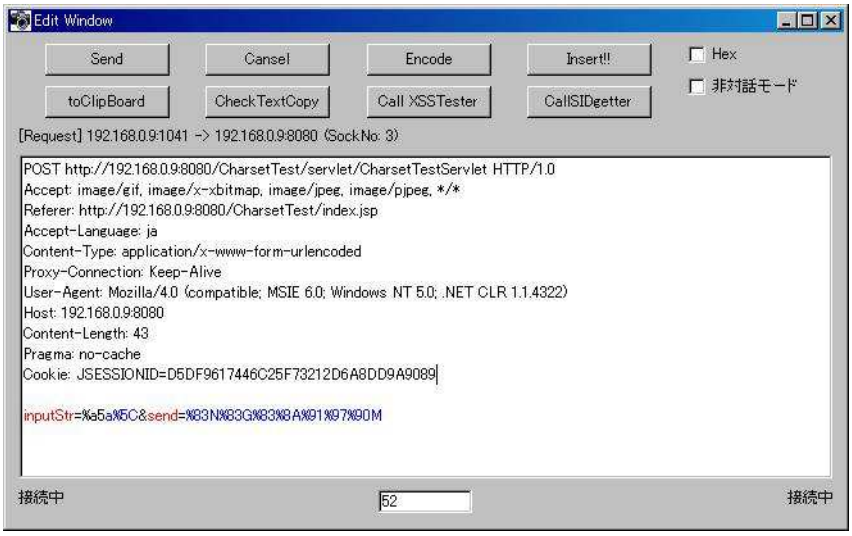

図4．1.4-9 : POST **されるデータを改変しテキストデータの先頭に「**%a5**」を付与してサーバへ送る**

図4．1.4-10：図 4．1.4-9の結果(Java_Windows-31J)

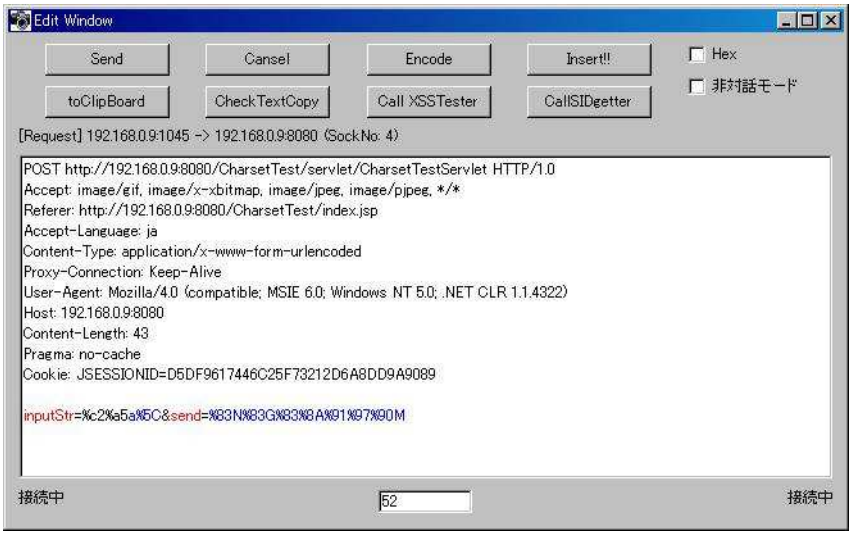

図4．1.4-11：POST **されるデータを改変しテキストデータの先頭に**

0xa5 **の** UTF-8 **表現である「**%c2%a5**」を付与してサーバへ送る**

図4．1.4-12：図 4．1.4-11の結果(Java_Windows-31J)

図4．1.4-13：QueryString **に「**%u00a5**」を付与した結果**(Java_Windows-31J)

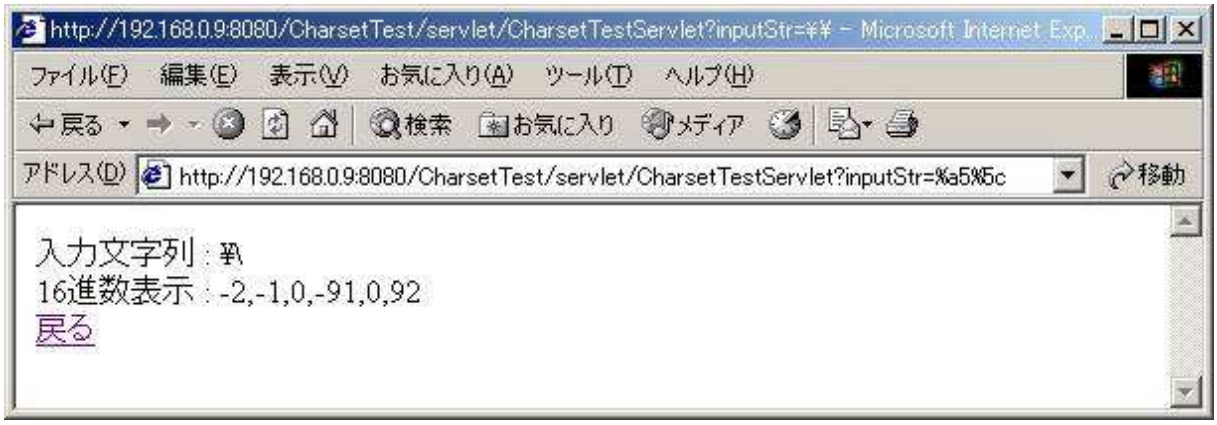

図4．1.4-14：QueryString **に「**%a5**」を付与した結果**(Java_Windows-31J)

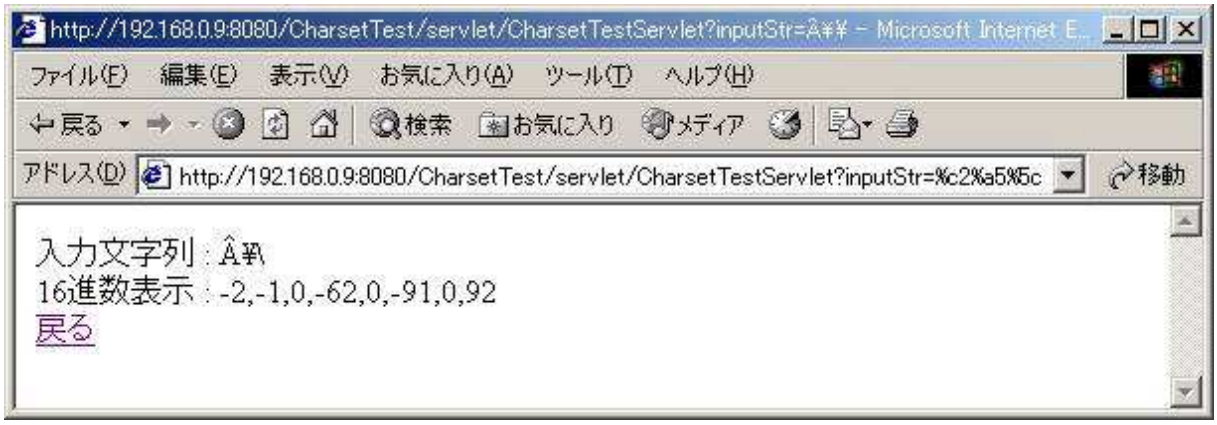

図4．1.4-15：QueryString **に** 0xa5 **の** UTF-8 **表現である「**%c2%a5**」を付与した結果**(Java_Windows-31J)

5　執筆者と更新履歴など

5.1 執筆者

- active@window.goukaku.com
- yuya.yamaguchi@gmail.com
- forcing@gmail.com

5.2 更新履歴

最初のバージョン : 2005 年 11 月 15 日

ver1.1 : **アップロードを忘れて未公開**

1.1で「衆知徹底」→「周知徹底」

6.7で「c:¥java¥a」→「c:¥z¥a」に修正

2.3, 6.2 を追加

ver1.2 : 2006 年 12 月 06 日

題名を変更

文章全体に渡っての誤字脱字、表現の修正

「Windows の LDAP インジェクションについて修正」(LDAP インジェクションに対しては問題はないため、削除した)

「4 アタックベクタ」を追加

3.5、3. 5.1、6.12 を追加

6.8 を追加

5.3 の URL が記述ミスだったのを修正

1.3 に「サニタイジングによる対策 2」を追加

5.3 本文書の最新バージョン

http://rocketeer.dip.jp/unicode/index.htm

6　付記

6.1 WindowsNT 系列での UNICODE 変換リストのプログラムについて

- UnicodeIsAll.bat

```
@FOR /L %%I IN (0,1,65535) DO @CALL UnicodeIsSub.bat %%I
```

- UnicodeIsSub.bat

```
@ECHO OFF
UnicodeIs.exe c %1
SET myANS=1
IF %ERRORLEVEL% == 0 SET myANS=0
IF %ERRORLEVEL% == 1 SET myANS=0
IF NOT %myANS%==0 ECHO %1
SET myANS=
```

- UnicodeIs.cpp

```
CWinApp theApp;
using namespace std;
int myMain(int hiKisu,char hiKisuC3,int inCode2,int inCode,int meate){
        unsigned char *p;
        int ansRet;
        unsigned char hako[6];
        unsigned char ansHako[6];
        int mojiSu;
        int ans;
        unsigned char c1;
        unsigned char c2;
        unsigned int ansInt;
        memset(hako,0x00,sizeof(hako));
        memset(ansHako,0x00,sizeof(ansHako));
        p = (unsigned char*)hako;
        ansRet = 0;
//      i = 0x2025;
```

```
		*(p+1) = (char)((int)(inCode/256));
		*p = (char)((int)(inCode%256));
		if(inCode2 != 0){
			*(p+3) = (char)(inCode2/256) + 220;
			*(p+2) = (char)(inCode2%256);
			*(p+1) = (char)((int)(*(p+1)) + 216);
		}
		mojiSu = WideCharToMultiByte(CP_ACP,NULL,(LPWSTR)p,-1,NULL,0,NULL,NULL);
		ans = WideCharToMultiByte(CP_ACP,NULL,(LPWSTR)p,-1,(char*)ansHako,sizeof(ansHako)-1,NULL,NULL);
		p = ansHako;
		c1 = *p;
		c2 = *(p+1);
		ansInt = 256 * (int)(c2) + (int)(c1);
		if(hiKisu == 2){
			if(2 < mojiSu){
				if(0 < (int)(c2) && (int)(c2) < 128 && (int)(c1) < 128){
					printf("%d(u%02x%02x)
- %d(0x%02x)[%c]¥n",inCode,(int)(inCode/256),(int)(inCode%256),ansInt,ansInt,*ansHako);
					ansRet = 1;
				}
			}
		}else{
			if(ansInt == meate){
				if(3 < hiKisu && hiKisuC3 == 'v'){
					if(inCode2 != 0){

		printf("%d,%d(u%02x%02x,%02x%02x)",inCode2,inCode,(int)(inCode2/256),(int)(inCode2%256),(int)(inCode/256),(i
nt)(inCode%256));
					}else{

		printf("%d(u%02x%02x)",inCode,(int)(inCode/256),(int)(inCode%256));
					}
					printf(" - %d(0x%02x)[%c]¥n",ansInt,ansInt,*ansHako);
				}
				ansRet = 1;
			}
```

```
	}
	return ansRet;
}
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[]){
	int nRetCode = 0;
	// MFC の初期化および初期化失敗時のエラーの出力
	if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0)){
		cerr << _T("Fatal Error: MFC initialization failed") << endl;
		nRetCode = -1;
	}else{
		unsigned int meate = 0;
		unsigned long i;
		unsigned long iMax;
		unsigned long iMin;
		unsigned long j;
		unsigned char c='a';
		int ansI = 0;
		iMax = 65535;
//		iMax = 25;
		iMin = 0;
//		iMin = 20;
			// 引数チェック
		if(1 < argc){
			if(2 < argc){
				meate = (unsigned long)atol(argv[2]);
				if(3 < argc && *argv[3] == 'v'){
					printf("meate : %d(0x%02x)¥n",meate,meate);
					c = *argv[3];
				}
			}
			i=iMin;
			for(i=iMin;i<=iMax;i++){
				ansI = ansI + myMain(argc,c,0,i,meate);
			}
			for(j=1;j<1024;j++){
				for(i=0;i<1024;i++){
```

```
                                        ansI = ansI + myMain(argc,c,j,i,meate);
                                }
                        }
                        nRetCode = ansI;
                        if(3 < argc && *argv[3] == 'v'){
                                printf("Count=%d¥n",ansI);
                        }
                }else{
                        printf("ASCII コードを指定して、重複する UNICODE 文字を探す旅 ver1.0¥n");
                        printf("usage: %s c ASCIIcode v¥n",argv[0]);
                        printf("       %s c ASCIIcode¥n",argv[0]);
                        printf("       %s a¥n",argv[0]);
                }
        }
        return nRetCode;
}
```

6.2 Java での UNICODE 変換リストのプログラムについて

```
import java.io.UnsupportedEncodingException;
import org.ingrid.util.UnicodeTool;

public class UnicodeTest {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java -jar UnicodeTest.jar <encoding>");
            System.out.println("<encoding> is EUC-JP, Shift_JIS etc..");
            System.exit(1);
        }

        /*
         * Unicode => 他の文字コード変換マップ配列
         */
        byte[][] array;
```

```
array = getByteArray(args[0]);

/*
 * Ascii 文字の範囲(0〜0x7f)で繰り返し検索
 */
for (int i = 0; i < 0x80; i++) {
        // 重複カウント
    int count = 0;
    // 出力用 StringBuffer
    StringBuffer sb = new StringBuffer();

    for (int j = 0; j < 0x10000; j += 0x0001) {
        /*
         * 変換後の byte の長さが 1 で、
         * 変換後の byte 値が 0x3f でなくて、
         * 変換後の byte 値が現在検索中の Ascii 文字と
         * かぶる場合には、count + 1
         */
        if ((array[j].length == 1)
                && (array[j][0] != 0x3f)
            && (array[j][0] == i)) {

            sb.append("Unicode : ¥¥u"
                    + UnicodeTool.toHexString((char) j) + "¥n");
            sb.append("文字表示: " + (char) j + "¥n");
            sb.append("変換 byte: " + UnicodeTool.toHexString(array[j])
                    + "¥n");
            sb.append("¥n");
            count += 1;
        }
    }

    /*
     * count = 1 は、1 つの Unicode の変換の結果
     * 1 つの Ascii コードに変換されているものが該当
     * (Unicode と Ascii コードの重なっている部分)
```

```
                * count >= 2 の時は、複数の Unicode が 1 つの Ascii コード
                * に変換されているものが該当(Unicode バグの巣となる可能性)
                */
               if (count > 1) {
                    System.out.print(sb.toString());
               }
          }
     }

     /*
      * 変換マップ 2 次元配列を作る
      * array[2345]には¥u2345 の変換結果の byte[]が入る
      *
      */
     private static byte[][] getByteArray(String encoding) {
          byte[][] array = new byte[65536][];
          try {
               for (int i = 0x0000; i < 0x10000; i += 0x0001) {
                    String s = String.valueOf((char) i);
                    byte[] ba;
                    ba = s.getBytes(encoding);
                    array[i] = ba;
               }
          } catch (UnsupportedEncodingException e) {
               e.printStackTrace();
               System.exit(1);
          }
          return array;
     }
}
```

実行結果

C:¥cygwin¥home¥Owner¥unicodetest>java -version

java version "1.5.0_05"

Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_05-b05)

Java HotSpot(TM) Client VM (build 1.5.0_05-b05, mixed mode, sharing)

C:¥cygwin¥home¥Owner¥unicodetest>java -jar UnicodeTest.jar Shift_JIS

Unicode : ¥u005c

文字表示: ¥

変換 byte: 5c

Unicode : ¥u00a5

文字表示: ¥

変換 byte: 5c

Unicode : ¥u007e

文字表示: ~

変換 byte: 7e

Unicode : ¥u203e

文字表示: ~

変換 byte: 7e

6.3 tchar.h を使って、引数の hex 表示をする VC++6.0SP6 プログラム

テンプレート : Win32ConsoleApplication → MFC を使った Hello プログラム

// argvWin.cpp : コンソール アプリケーション用のエントリ ポイントの定義

#include "stdafx.h"

#include "argvWin.h"

#ifdef _DEBUG

#define new DEBUG_NEW

```
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
//////////////////////////////////////////////////////////////////////////
// 唯一のアプリケーション オブジェクト
CWinApp theApp;
using namespace std;
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[]){
	int nRetCode = 0;
	int i;
	int j;
	int jLen;
	TCHAR tc;
	TCHAR *tcp;
	char *pp;
	int bai;

	// MFC の初期化および初期化失敗時のエラーの出力
	if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0)){
		cerr << _T("Fatal Error: MFC initialization failed") << endl;
		nRetCode = 1;
	}else{
		bai = 1;
		#ifdef _UNICODE
			bai = 2;
			printf("unicode¥n");
		#endif

		#ifdef _MBCS
			bai = 1;
			printf("mbcs¥n");
		#endif

		for(i=0;i<argc;i++){
			jLen = lstrlen(argv[i]);
			_tprintf(_T("%d : "),i);
```

```
                tcp = argv[i];
                for(j=0;j<jLen;j++){
                        tc = *tcp;
                        _tprintf(_T("%x"),tc);
                        tcp++;
                }
                printf("     : ");
                pp = (char*)argv[1];
                jLen = bai * jLen;
                for(j=0;j<jLen;j++){
                        printf("%x",(char)(*pp));
                        pp++;
                }
                _tprintf(_T("¥n"));
}               }               return nRetCode;        }
```

6.4 Variant 型の引数を ANSI で受け取るメソッドと Unicode(Binary) で受け取るメソッドのある COM(MS-VC++6.0SP6)

テンプレート : ATL-COM Wizard → ATL クラス(シンプルオブジェクト)

```
// UnicodeTest.cpp : CUnicodeTest のインプリメンテーション
#include "stdafx.h"
#include "UnicodeCom.h"
#include "UnicodeTest.h"
#include <comdef.h>
/////////////////////////////////////////////////////////////////////////////
// CUnicodeTest
STDMETHODIMP CUnicodeTest::testANSI(VARIANT vin){
        _variant_t myVariant;
        _bstr_t myBSTR;
        unsigned long myLen;
        unsigned char *cp;
        char *pp;
```

```
        unsigned char *p;
        unsigned char c;

                // 自分のバリアント型へコピー
        myVariant = _variant_t(&vin);
                // BSTR 型へ変換
        myVariant.ChangeType(VT_BSTR,NULL);
                // 自分の BSTR 型へコピー
        myBSTR = _bstr_t(myVariant);
                // 自分の Variant 型を解放
        myVariant.Clear();
                // BSTR 型の文字長はいくつですか?
        myLen = (unsigned long)myBSTR.length();
                // UNICODE も考えて、2 倍のメモリを確保
        myLen = 2* myLen + 2;
        cp = (unsigned char*)malloc(myLen);
        memset(cp,0x00,myLen);
        myLen -= 2;
                // 確保したメモリへコピー
        pp = (char*)myBSTR;
        strncpy((char*)cp,pp,myLen);
                // NULL まで Hex で表示
        p = cp;
        while(*p != '¥0'){
                c = *p;
                if((int)c < 10){
                        printf("0");
                }
                printf("%x",c);
                p++;
        }
        printf("¥n");
                // メモリ解放
        free(cp);
        return S_OK;
}
```

```
STDMETHODIMP CUnicodeTest::testUNICODE(VARIANT vin){
	_variant_t myVariant;
	unsigned long myLen;
	unsigned char *cp;
	unsigned char *p;
	unsigned char c;
	unsigned long i;

		// 自分のバリアント型へコピー
	myVariant = _variant_t(&vin);
		// unsigned int の SAFEARRAY 配列へ変換
	myVariant.ChangeType(VT_ARRAY | VT_UI1,NULL);
		// 配列長はいくつですか?
	myLen = (unsigned long)myVariant.parray->rgsabound->cElements;
		// メモリを確保してコピーする
	myLen++;
	cp = (unsigned char*)malloc(myLen);
	memset(cp,0x00,myLen);
	myLen -= 1;
	memcpy((char*)cp,(char*)myVariant.parray->pvData,myLen);
		// 自分の Variant 型を解放
	myVariant.Clear();
		// 長さ分 hex 表示
	// NULL まで Hex で表示
	p = cp;
	for(i=0;i<myLen;i++){
		c = *p;
		if((int)c < 10){
			printf("0");
		}
		printf("%x",c);
		p++;
	}
	printf("¥n");
		// メモリ解放
```

```
        free(cp);
        return S_OK;
}
```

6.5 「6.4」の COM を呼び出すテストスクリプト

```
Set obj = WScript.CreateObject("UnicodeCom.UnicodeTest")

str = "abc あ xyz"
WScript.Echo "Script Start"
WScript.Echo "target string : " & str

WScript.Echo "TestAnsi exec"
obj.testANSI(str)

WScript.Echo "TestUnicode exec"
obj.testUNICODE(str)

WScript.Echo "Script End"
Set obj = Nothing
WScript.QUit
```

6.6 Java で書かれたファイルアクセスプログラム

あらかじめ「c:¥java¥a」というディレクトリを作成しておく。

```
        import java.lang.*;
        import java.io.*;

        class JavaFileTest{
                public static void main(String[] args){
                        Character ch = new Character('¥u00a5');
                        String chStr = ch.toString();
```

```
                String str;
                String str1 = "Hello";
                File myFileClass;
                FileOutputStream fOut;
                if(args.length == 0){
                        str = "c:¥¥java¥¥a" + "¥¥" + "test.txt";
                }else{
                        str = "c:¥¥java¥¥a" + chStr + "test.txt";
                }
                System.out.println("FilePath : " + str);
                myFileClass = new File(str);
                try{
                        fOut = new FileOutputStream(myFileClass);
                        try{
                                fOut.write(str1.getBytes());
                                fOut.close();
                        }catch(IOException e){
                        }
                }catch(FileNotFoundException e){
                }
        }
}
```

6.7 Python で書かれたファイル読み出しプログラム

あらかじめ「c:¥z¥a」というディレクトリを作成しておく。

```
import os


yen = u'C:¥¥z¥¥a¥u00a5unicode.txt'
f = open(yen, 'a')
f.write('test')
f.close()
```

6.8 VB の Open ステートメントと Scripting.SystemFileObject オブジェクトを使ったファイル読み出しプログラム (VB6SP6)

図6.8-1 : GUI **画面**

ラジオボタンで、ファイル読み出しとして Open **ステートメントを使うか**

Scripting.FileSystemObject **オブジェクトを使うかを決定する**

チェックボックスで、右二段目のテキストボックスの「¥」を「0xa5**」に置換してからファイルパスを生成する**

「ファイルパス」=「最上段」+「右二段目」

```
Option Explicit

Private Sub Command1_Click()
    End
End Sub

Private Sub Command2_Click()
    Dim myPath As String
    Dim myByte1() As Byte
    Dim dispString As String
    Dim i As Long
    Dim FSObj As Object
    Dim FileObj As Object
    Dim iMax As Long

    myPath = Text2.Text
```

```
dispString = ""
myByte1 = myPath
iMax = UBound(myByte1)
For i = 0 To iMax
    dispString = dispString & "(" & CStr(myByte1(i)) & ")"
Next
Label1.Caption = dispString

If Check1.Value = 1 Then
    For i = 0 To iMax
        If myByte1(i) = 92 Then
            myByte1(i) = 165
        End If
    Next
    myPath = myByte1
End If
dispString = ""
myByte1 = myPath
iMax = UBound(myByte1)
For i = 0 To iMax
    dispString = dispString & "(" & CStr(myByte1(i)) & ")"
Next
Label2.Caption = dispString

If Option1(0).Value = True Then
    Set FSObj = CreateObject("Scripting.FileSystemObject")
    Set FileObj = FSObj.CreateTextFile(Text1.Text & "¥" & myPath)
    FileObj.Write "Hello"
    FileObj.Close
    Set FileObj = Nothing
    Set FSObj = Nothing
Else
    Open Text1.Text & "¥" & myPath For Output As #1
    Close #1
End If
```

```
    MsgBox "ファイルできた?", vbOKOnly, "OK"
End Sub

Private Sub Command3_Click()
    Dim str As String
    Dim ret As Variant
    str = "本プログラムは、UNICODE バグを用いた「¥」の迂回可能性をテストするプログラム" & vbCrLf
    str = str & "VB には、open ステートメントによるファイル読み書きと、" & vbCrLf
    str = str & "COM の Scripting.FileSystemObject を使ったファイルアクセス方法がある(こちら側は ActiveScript でも使う)" & vbCrLf
    str = str & "という事で、ファイル名に「¥」を入れるとそれより前は「ディレクトリ」になる" & vbCrLf
    str = str & "んで「UNICODE Bug」チェックをするとどうなるか?" & vbCrLf
    str = str & "もう一つの「¥」がデリミタとして解釈されなければ、セキュリティ問題はなし" & vbCrLf
    str = str & "解釈されれば、危険性あり。" & vbCrLf
    str = str & "という事になるよ〜。"
    ret = MsgBox(str, vbOKOnly, "Help")
End Sub

Private Sub Form_Load()
    Dim str As String
    Text1.Text = App.Path & "¥"
End Sub

Private Sub Text1_OLEDragDrop(Data As DataObject, Effect As Long, Button As Integer, Shift As Integer, X As Single, Y As Single)
    Text1.Text = Data.Files.Item(1) & "¥"
End Sub
```

6.9 WSH の Run メソッドでコマンド実行するスクリプト

```
Option Explicit
Dim WshObj
Dim ArgObj
```

```
Dim str

Set WshObj = WScript.CreateObject("WScript.Shell")
Set ArgObj = WScript.Arguments

REM 0x7c = 124 [|]
rem str = chrW(124)
rem str = chrW(166)
rem WScript.Echo str

WScript.Echo ArgObj.Count

If 0 < ArgObj.Count Then
 Select Case CStr(ArgObj(0))
   Case "1"
    WshObj.Run "a.exe | b.exe"
   Case "2"
    WshObj.Run "cmd.exe /c a.exe | b.exe"
   Case "3"
    WshObj.Run "cmd.exe /c a.exe " & chrW(124) & " b.exe"
   Case "4"
    WshObj.Run "cmd.exe /c a.exe " & chrW(166) & " b.exe"
 End Select
Else
 WScript.Echo "Read Sourcecode"
End If

Set WshObj = Nothing
WScript.Quit
```

6.10 WSH の Exec メソッドでコマンド実行するスクリプト

```
Option Explicit
Dim WshObj
```

```
Dim ArgObj
Dim ExecObj
Dim str

Set WshObj = WScript.CreateObject("WScript.Shell")
Set ArgObj = WScript.Arguments

REM 0x7c = 124 [|]
rem str = chrW(124)
rem str = chrW(166)
rem WScript.Echo str

WScript.Echo ArgObj.Count

If 0 < ArgObj.Count Then
 Select Case CStr(ArgObj(0))
  Case "1"
   str = "a.exe | b.exe"
  Case "2"
   str = "cmd.exe /c a.exe | b.exe"
  Case "3"
   str = "cmd.exe /c a.exe " & chrW(124) & " b.exe"
  Case "4"
   str = "cmd.exe /c a.exe " & chrW(166) & " b.exe"
 End Select
 Set ExecObj = WshObj.Exec(str)
 While ExecObj.Status = 0
  WScript.Sleep 100
 Wend
 str = ""
 While Not ExecObj.StdOut.AtEndOfStream
  str = str & ExecObj.StdOut.Read(1)
 Wend
 Set ExecObj = Nothing
 WScript.Echo str
Else
```

```
 WScript.Echo "Read Sourcecode"
End If

Set WshObj = Nothing
WScript.Quit
```

6.11 外部コマンド呼び出しによって呼び出されるプログラム

本プログラムが実行されると、実行プログラムと同一ディレクトリに、「実行プログラム」+「.txt」というファイルが作成される。

```
/* テストプログラム */
/* プログラム名 +  */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc,char *argv[]){
    unsigned char *filename;
    unsigned char *p;
    unsigned long filenameLen;
    FILE *fp;
    filenameLen = strlen(argv[0]);
    filename = (unsigned char*)malloc(filenameLen+5);
    memset(filename,0x00,filenameLen+5);

            /* get filename */
    strcpy(filename,argv[0]);
    p = filename + filenameLen;
    strcpy(p,".txt");

            /* write file */
```

```
    fp = fopen(filename,"w");
    fprintf(fp,"Hello");
    fclose(fp);

                /* ending */
    printf("write %s¥n",filename);

    return 0;
}
```

6.12 MS-XML コアサービスの COM オブジェクトを使った XML 文書検索プログラム (VB6SP6)

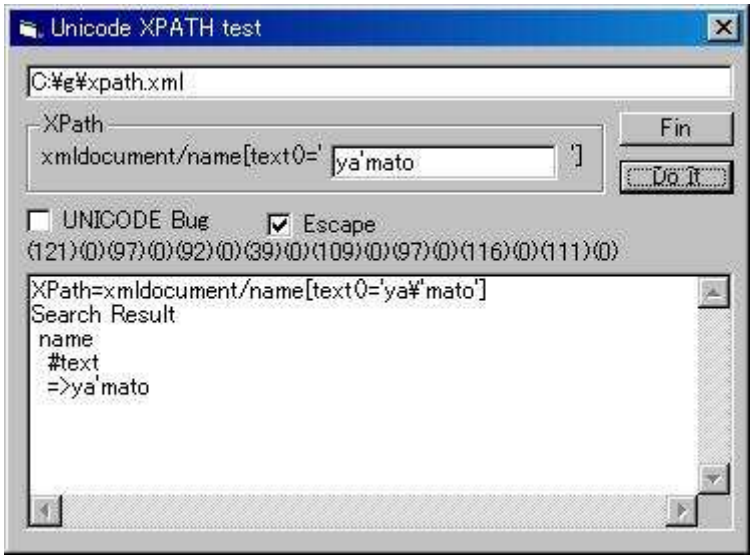

図6.12-1 : GUI **画面**

最上段のテキストボックスは、読み込む XML **文書のファイルパス**

「UNICODE Bug**」チェックボックスで、**XPath **検索条件中の「¥」を「**0xa5**」に置換してから検索**

「Escape**」チェックボックスでは「¥」→「¥¥」、「'」→「¥'」にエスケープする**

最下段のテキストボックスは、検索結果

```
Option Explicit
```

```
Private Sub Command1_Click()
    Dim xmlRootObj As Object
    Dim XMLDocObj As Object
    Dim XMLDocChildObj As Object
    Dim iMax As Long
    Dim i As Long
    Dim jMax As Long
    Dim j As Long
    Dim str As String
    Dim obj As Object
    Dim myObj As Object
    Dim myByte() As Byte

    Rem ここに XPATH を書く
    Rem str = "xmldocument/BB/name"
    Rem str = "xmldocument/BB/name[text()='ya¥¥ma¥'to']"
    str = Text3.Text
    Set xmlRootObj = CreateObject("Microsoft.XMLDom")
    xmlRootObj.Load Text1.Text
    xmlRootObj.async = True

    Rem 全部取得
    Rem Set XMLDocObj = xmlRootObj.childNodes.Item(1).ChildNodes
    Rem 検索して取得
    myByte = str
    iMax = UBound(myByte)
        Rem XPATH 文字列の Hex 表示
    Label1.Caption = ""
    For i = 0 To iMax
        Label1.Caption = Label1.Caption & "(" & CStr(CInt(myByte(i))) & ")"
    Next
        Rem 92-> 165
    If Check1.Value = 1 Then
        Label1.Caption = ""
        For i = 0 To iMax
```

```
            If myByte(i) = 92 Then
                myByte(i) = 165
            End If
            Label1.Caption = Label1.Caption & "(" & CStr(CInt(myByte(i))) & ")"
        Next
        str = myByte
    End If
    If Check2.Value = 1 Then
        str = Replace(str, "¥", "¥¥")
        str = Replace(str, """, "¥"")
        Label1.Caption = ""
        myByte = str
        iMax = UBound(myByte)
        For i = 0 To iMax
            Label1.Caption = Label1.Caption & "(" & CStr(CInt(myByte(i))) & ")"
        Next
    End If
    str = Label2.Caption & str & Label3.Caption
    Set XMLDocObj = xmlRootObj.selectNodes(str)

        Rem 再帰処理
    Text2.Text = "XPath=" & str & vbCrLf & "Search Result"
    XMLChild XMLDocObj, "", 0
        Rem 終了処理
    Set XMLDocChildObj = Nothing
    Set XMLDocObj = Nothing
    Set xmlRootObj = Nothing
End Sub

Private Sub Command2_Click()
    End
End Sub

Private Sub XMLChild(iObj As Object, iStr As String, mode As Long)
    Dim myObj As Object
    Dim iMax As Long
```

```
    Dim i As Long
    If mode = 0 Then
        Set myObj = iObj
    Else
        Text2.Text = Text2.Text & vbCrLf & iStr & iObj.NodeName
        Set myObj = iObj.ChildNodes
    End If
    iMax = myObj.length
    If 0 < iMax Then
        iMax = iMax - 1
        For i = 0 To iMax
            XMLChild myObj.Item(i), iStr & " ", 1
        Next
    Else
        Text2.Text = Text2.Text & vbCrLf & iStr & "=>" & GetValue(iObj)
    End If
    Set myObj = Nothing
End Sub

Private Function GetValue(iObj) As String
    On Error Resume Next
    Dim ans As String
    ans = "(null)"
    ans = CStr(iObj.nodeValue)
    GetValue = ans
End Function

Private Sub Form_Load()
    Text1.Text = App.Path & "¥xpath.xml"
End Sub
```

6.13 IIS-ASP の文字列データ(String in Variant)のバイト列表示(10進)(ActiveX DLL by VB6SP6)

オブジェクト「HexDispCom.Class」の「GetHexDisp」メソッドのソースコード

```
    Rem Hex 表示というより 10 進数表示
Public Function GetHexDisp(inputStr As Variant) As Variant
    Dim myByte() As Byte
    Dim str As String
    Dim ans As String
    Dim i As Long
    Dim iMax As Long
    ans = ""
    str = CStr(inputStr)
    myByte = str
    iMax = UBound(myByte)
    For i = 0 To iMax
        ans = ans & "(" & CStr(CInt(myByte(i))) & ")"
    Next
    GetHexDisp = ans
End Function
```

6.14 IIS-ASP で Web ブラウザから受け取ったリクエストのバイト列表示(IIS-ASP + VBScript)

非 UNICODE バージョン (test.asp)

```
<%
 Option Explicit
 Dim str
 Dim ans
 Dim obj
 str = Request.QueryString("inputStr")
```

```
 Set obj = Server.CreateObject("HexDispCom.Class")
 ans = obj.GetHexDisp(str)
 Set obj = Nothing
%>
<html>
 <head><meta http-equiv="Content-Type" content="text/html; charset=UTF-8" /></head></body>
<%
 Response.Write str & "<br>" & ans
%>
</body></html>
```

UNICODE 許可バージョン (testu.asp)

```
<%@ LANGUAGE=VBScript CODEPAGE=65001%>
<%
 Option Explicit
 Dim str
 Dim ans
 Dim obj
 str = Request.QueryString("inputStr")
 Set obj = Server.CreateObject("HexDispCom.Class")
 ans = obj.GetHexDisp(str)
 Set obj = Nothing
%>
<html>
 <head><meta http-equiv="Content-Type" content="text/html; charset=UTF-8" /></head></body>
<%

 Response.Write str & "<br>" & ans
%>
</body></html>
```

6.15 ASP.Net で Web ブラウザから受け取ったリクエストのバイト列表示(C#)

(HexDispCS.aspx)

.NET Framework は内部 UNICODE(UTF-16)なので、バイト列を取得するときに、エンコード方式を「UTF-16」にすることで、内部状態を壊さないようにしている

```
<%@ Page Language="C#" %>
<script runat="server">

    // ページのコードをここに記述してください。
    //

    void Button1_Click(object sender, EventArgs e) {
        String myStr;
        String myHexStr;
        myStr = TextBox1.Text;

        Encoding EncodeObj = Encoding.GetEncoding("utf-16");
        byte []myByteHako = EncodeObj.GetBytes(myStr);
        myHexStr = BitConverter.ToString(myByteHako);

        Label3.Text = myStr;
        Label1.Text = myHexStr;
    }

</script>
<html>
<head>
</head>
<body>
    <form runat="server">
        <p>
            <asp:Label id="Label2" runat="server">入力されたデータを見てみる(unicode)</asp:Label>
        </p>
```

```
        <p>
            <asp:TextBox id="TextBox1" runat="server" Width="316px"></asp:TextBox>
        </p>
        <p>
            <asp:Button id="Button1" onclick="Button1_Click" runat="server" Text="Button"></asp:Button>
        </p>
        <p>
            <asp:Label id="Label3" runat="server">Label</asp:Label>
        </p>
        <p>
            <asp:Label id="Label1" runat="server">Label</asp:Label>
        </p>
        <!-- コンテンツをここに配置してください。 -->
    </form>
</body>
</html>
```

6.16 ASP.Net で Web ブラウザから受け取ったリクエストのバイト列表示(VB.NET)

(HexDispCS.aspx)

.NET Framework は内部 UNICODE(UTF-16)なので、バイト列を取得するときに、エンコード方式を「UTF-16」にすることで、内部状態を壊さないようにしている

```
<%@ Page Language="VB" %>
<script runat="server">

    ' ページのコードをここに記述してください。
    '

    Sub Button1_Click(sender As Object, e As EventArgs)
        Dim myStr As String
        Dim myHexStr As String
        Dim EncodeObj As Encoding
```

```
        Dim myByteHako() As byte
        myStr = TextBox1.Text

        EncodeObj = Encoding.GetEncoding("utf-16")
        myByteHako = EncodeObj.GetBytes(myStr)
        myHexStr = BitConverter.ToString(myByteHako)

        Label3.Text = myStr
        Label1.Text = myHexStr
    End Sub

</script>
<html>
<head>
</head>
<body>
    <form runat="server">
        <p>
            <asp:Label id="Label2" runat="server">入力されたデータを見てみる(unicode)</asp:Label>
        </p>
        <p>
            <asp:TextBox id="TextBox1" runat="server" Width="388px"></asp:TextBox>
        </p>
        <p>
            <asp:Button id="Button1" onclick="Button1_Click" runat="server" Text="Button"></asp:Button>
        </p>
        <p>
            <asp:Label id="Label3" runat="server">Label</asp:Label>
        </p>
        <p>
            <asp:Label id="Label1" runat="server">Label</asp:Label>
            <!-- コンテンツをここに配置してください。 -->
        </p>
    </form>
</body>
</html>
```

6.17　CGI として受け取ったデータのバイト列表示プログラム

MS-VisualStudio6.0 SP6 にて、MFC 付 Win32 Console としてプロジェクトを構成

```
// testCgi.cpp : コンソール アプリケーション用のエントリ ポイントの定義
#include "stdafx.h"
#include "testCgi.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
int isnumerics(char*);
//////////////////////////////////////////////////////////////////////////
// 唯一のアプリケーション オブジェクト
CWinApp theApp;
using namespace std;
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[]){
int nRetCode = 0;
// MFC の初期化および初期化失敗時のエラーの出力
if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0)){
        cerr << _T("Fatal Error: MFC initialization failed") << endl;
        nRetCode = 1;
}else{
        unsigned long ul;
        unsigned long i;
        unsigned long iMax;
        unsigned long contentLength;
        unsigned char *pHex;
        unsigned char c;
        TCHAR *tp;
        TCHAR *tp1;
        TCHAR *length_char = _T("CONTENT_LENGTH");
        TCHAR *query_string = _T("QUERY_STRING");
```

```
contentLength = 0;
        // HTTP ヘッダの送信
        // 基本は ANSI で出力
printf("Content-type: text/html¥n¥n");
if(sizeof(TCHAR) == 2){
        printf("UNICODE complie¥n");
}else{
        printf("ANSI complie¥n");
}
ul = 0;
printf("UNICODE Attack Vector Test¥n");
while(envp[ul] != NULL){
        tp = envp[ul];
                // あらかじめ、名前の先頭(tp)と値の先頭(tp1)を調べておく
        tp1 = tp;
        while(tp1 != NULL){
                if(_tcsncicmp(tp1,_T("="),1) == 0){
                        tp1++;
                        break;
                }
                tp1++;
        }
                // CONTENT_LENGTH と一致するかどうか?
                // 一致するなら、標準入力のデータ長として取得する
        if(_tcsncicmp(tp,length_char,14) == 0){     contentLength    = _ttol(tp1);   }
        if(_tcsncicmp(tp,query_string,12) == 0){
                        // 値の先頭のひとつ前(「=」のあるところ)に NULL を埋め込み
                        // 「名前」と「値」を分断する
                *(tp1-1) = NULL;
                _tprintf(_T("%s=%s¥n"),tp,tp1);
                iMax = _tcslen(tp1) * sizeof(TCHAR);
                pHex = (unsigned char*)tp1;
                for(i=0;i<iMax;i++){
                        c = *pHex;
                        printf("0x%x,",c);
                        pHex++;
```

```
                    }
                    printf("¥n");
                }
                ul++;
        }
        printf("¥nComandLine List¥n");
        ul = 0;
        if(0 < argc){
                for(ul=0;ul<argc;ul++){
                    printf("%d : ",ul);
                    _tprintf(_T("%s¥n"),argv[ul]);
                    iMax = _tcslen(argv[ul]) * sizeof(TCHAR);
                    pHex = (unsigned char*)argv[ul];
                    for(i=0;i<iMax;i++){
                        c = *pHex;
                        printf("0x%x,",c);
                        pHex++;
                    }
                    printf("¥n");
                }
        }
        if(contentLength != 0){
                printf("¥nStdin Data¥n");
                tp = (TCHAR*)calloc(contentLength+1,sizeof(TCHAR));
                if(tp == NULL){
                    printf("Memory Error");
                }else{
                    tp1 = tp;
                    for(ul=0;ul<contentLength;ul++){
                        *tp1 = _gettchar();
                        tp1++;
                    }
                    *tp1 = '¥0';
                    _tprintf(_T("%s¥n"),tp);
                    iMax = contentLength * sizeof(TCHAR);
                    pHex = (unsigned char*)tp;
```

```
                                        for(i=0;i<iMax;i++){
                                                c = *pHex;
                                                printf("0x%x,",c);
                                                pHex++;
                                        }
                                        free(tp);               }                               }
                printf("¥n");           }
return nRetCode;        }
```

6.18 JavaServler で Web ブラウザから受け取ったリクエストのバイト列表示

方式を「UTF-16」にすることで、内部状態を壊さないようにしている

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
        // QueryString の場合は、doPost() ではなくて doGet() にするだけ
        String myStr;
        String myHexStr;
        int i;
        int iMax;
        int myI;
        myHexStr = "";
//      request.setCharacterEncoding("Windows-31J");
//      request.setCharacterEncoding("utf-8");          // 上の行とどちらか一方のみコメントアウトをはずす
        myStr = request.getParameter("inputStr");
        try{
                byte myByte[] = myStr.getBytes("UTF-16");
                iMax = myByte.length;
                for(i=0;i<iMax;i++){
                        myI = (int)myByte[i];
                        if(myHexStr != ""){
                                myHexStr += ",";
```

```
                                        }
                                        myHexStr += myI;
                                }
                }catch(Exception e){                    }
                myHexStr = HTMLEscape(myHexStr);
                myStr = HTMLEscape(myStr);
                request.setAttribute("inputStr",myStr);
                request.setAttribute("inputHexStr",myHexStr);
                getServletConfig().getServletContext().getRequestDispatcher("/result.jsp").forward(request,response);
}

private String HTMLEscape(String str){
                String tmp = str.replaceAll("&","&amp;");
                tmp = tmp.replaceAll("<","&lt;");
                tmp = tmp.replaceAll(">","&gt;");
                tmp = tmp.replaceAll("¥"","&quot;");
                return tmp.replaceAll("'","&#39;");
}
```

以　　上