CMP
United Business Media

Internet Programming

Drawing on GDI+ From Native C++

OCT 2003

windows::developer NETWORK

APPLICATION DEVELOPMENT FROM WINDOWS TO WEB

URL Canonicalization Testing

Extending ASP.NET Apps using HttpHandlers

Licensing in .NET

Creating Custom Columns for the WinForms DataGrid Control

Shared Data Segments

Preventing Class Derivation in Visual C++ .NET

Still More Do/While Macros

Volume 2 / No. 10

www.windevnet.com

Improve your knowledge of the latest technologies

Over 80 Sessions and Full-Day-Seminars
World's leading Windows speakers
Technical chairman Jeffrey Richter

www.winsummit.com

WinSuMMit 2003

European Windows Developers' Summit

Conference Centre Davos, Switzerland
September 29 to October 3, 2003

Keep yourself informed

☐ Yes, I would like to attend WinSummit 2003
☐ Please send me more information about WinSummit

If you have any questions or suggestions, don't hesitate to contact us: www.winsummit.com, Fax +41 1 881 44 95, Tel. +41 1 881 44 96

First Name ______
Last/Family Name ______
Job Title ______
Department ______
Company Name ______
Street Address ______
Town/City ______
Postal Code ______
Province/Country ______
Phone ______
Fax ______
E-Mail ______

owned by: DIGICOMP® EXPERT SEMINARS

Why you should attend

- Improve your knowledge of the latest technologies
- Discuss your technical questions with the world's top experts
- Obtain practical examples on the conference CD
- Broaden your horizon about future Microsoft Technologies and strengthen your personal market value
- Network with experts and other participants

Speakers at WinSummit

Francesco Balena	Steven Pratchner
Jason Clark	Jeff Prosise
Peter DeBetta	Brent Rector
Dino Esposito	Jeffrey Richter
Tim Ewald	John Robbins
Eric Meijer	etc
Adam Nathan	

For sponsor opportunities
please contact info@edpp.ch
or call +41 1 881 44 96

Who should attend

- Professional developers who
  - want to have an in-depth understanding of the mechanisms and the use of the .NET and web technologies
  - know the difference of learning from real experts
  - like to consolidate their knowledge through personal contact with their favourite experts
  - fear marketing sessions but enjoys in-depth technical content
  - are facing all aspects of modern Windows and web application development
  - wish to understand and use the best HOW-TOs to ensure the performance and scalability of their applications
  - want to have answers for their questions about the smartest use of Microsoft's technologies using C#, Visual Basic, Visual C++ with Visual Studio.NET
- Consultants who wish to improve their understanding of Microsoft technologies
- UNIX/Linux developers who want to catch up with Microsoft technologies

windows::developer
APPLICATION DEVELOPMENT FROM WINDOWS TO WEB NETWORK

PUBLISHER
Kerry Gates

EDITOR IN CHIEF
John Dorsey

BPA

EDITORIAL

MANAGING EDITOR **Amy Stephens**

CONTRIBUTING EDITORS **Dino Esposito, George Frazier, Richard Grimes, Petter Hesselberg, Paula Tomlinson, Victor R. Volkman**

EDITORIAL ADVISORY BOARD **Mark Baker, Dino Esposito, George Frazier, Richard Grimes, Petter Hesselberg, Mark Nelson, Mark Russinovich, Paula Tomlinson, Victor R. Volkman**

ASSOCIATE EDITOR **Della Song**

ART DIRECTOR **Beatriz Américo**

WEBMASTER **Joe Lucca**

SEND READER MAIL TO: **wdletter@cmp.com**

SUBSCRIPTION INQUIRIES: **wdnetwork@halldata.com**

ADVERTISING AND MARKETING

DIRECTOR OF SALES **David Timmons**

REGIONAL MANAGER, EAST
Jon Hampson 603-924-8500 *jhampson@cmp.com*

REGIONAL MANAGER, CENTRAL/SOUTHEAST
Ed Day 785-838-7547 *eday@cmp.com*

REGIONAL MANAGER, WEST
Michele Hurabiell 415-947-6199 *mhurabiell@cmp.com*

ACCOUNT MANAGER, ALL REGIONS
Julie Thibault 603-924-8400 *jthibault@cmp.com*

PRODUCTION COORDINATOR
Michael Penne *mpenne@cmp.com*

DIRECTOR OF MARKETING **Karen Tom**

CIRCULATION

SENIOR CIRCULATION MANAGER **Cherilyn Olmsted**

ASSISTANT CIRCULATION MANAGER **Gwen Olson**

SUBSCRIPTIONS: Annual renewable print subscriptions to *Windows Developer Network* are $34.99 U.S., $45 Canada and Mexico, $65 elsewhere. Payments must be made in U.S. dollars. Make checks payable to *Windows Developer Network*.

CUSTOMER SERVICE: For subscription orders and questions, contact lawrencecs@cmp.com.

ADVERTISING: For rate cards or other information on placing advertising in *Windows Developer Network*, contact the advertising department at 785-838-7500, or write *Windows Developer Network*, 4601 West 6th Street, Suite B, Lawrence, KS 66049 USA.

Windows Developer Network (ISSN 1543-6462) is published monthly by CMP Media LLC, 600 Harrison St., San Francisco, CA 94107 USA, 415-947-6000.

CMP MEDIA LLC

CORPORATE
PRESIDENT AND CEO **Gary Marshall**
EXECUTIVE VICE PRESIDENT AND CFO **John Day**
EXECUTIVE VICE PRESIDENT AND COO **Steve Weitzner**
EXECUTIVE V.P., CORPORATE SALES AND MARKETING **Jeff Patterson**
CHIEF INFORMATION OFFICER **Mike Mikos**
SENIOR V.P., OPERATIONS **Bill Amstutz**
SENIOR V.P., HUMAN RESOURCES **Leah Landro**
VICE PRESIDENT AND GENERAL COUNSEL **Sandra Grayson**

MARKET GROUPS
PRESIDENT, TECHNOLOGY SOLUTIONS **Robert Faletra**
PRESIDENT, HEALTHCARE MEDIA **Vicki Masseria**
V.P., GROUP PUBLISHER APPLIED TECHNOLOGIES **Philip Chapnick**
V.P., GROUP PUBLISHER INFORMATIONWEEK MEDIA NETWORK **Michael Friedenberg**
V.P., GROUP PUBLISHER ELECTRONICS **Paul Miller**
V.P., GROUP PUBLISHER NETWORK COMPUTING MEDIA NETWORK **Fritz Nelson**
V.P., GROUP PUBLISHER SOFTWARE DEVELOPMENT MEDIA **Peter Westerman**
CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT **Shannon Aronson**
CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT **Michael Zane**
CORPORATE DIRECTOR, PUBLISHING SERVICES **Marie Myers**

CMP
United Business Media

www.**windevnet**.com

CONTENTS

OCT 2003

windows::developer NETWORK VOL 2 • NO. 10

FEATURES

COLUMNS

VISIT US ONLINE: www.windevnet.com

DEPARTMENTS

PDF EXTRA

| Download code › **windevnet.com/wdn/code/** |

COMING NEXT MONTH: **DISTRIBUTED COMPUTING**

THE CONVERGENCE OF TELEPHONY and traditional desktop-based computer systems is an important market for just about every major software tool vendor. Automatic Speech Recognition (ASR) and Text-To-Speech (TTS) technologies have been available on desktop systems for several years, and both have steadily improved to make speech technology more than just a curiosity. Accessing data through Vocal User interfaces (VUIs) has the potential to grow into a $4 billion market over the next four years.

Microsoft is addressing that convergence with several server and client technologies. It has also entered into some key partnerships with companies already established in speech technologies including Intel, Intervoice, and SpeechWorks (now part of ScanSoft). This summer, Microsoft released the Microsoft Speech Server (MSS) beta 1, and the third beta of its Speech Application Software Development Kit (SASDK). More information about MSS is available at http://www.microsoft.com/speech/. Microsoft has designed its platform for a variety of different scenarios: Telephony-only apps, voice-enabled browser-based apps, and multimodal apps spanning both phone and web.

The telephony scenario is a voice-only system typically used for call centers, directory listings, or voice mail. In this set up, the MSS platform is integrated with an Intel Dialogic card in the server that ties into the company PBX. Intervoice is building the driver to talk to the Intel card. The browser-based scenario involves speech-enabling web apps so that users can opt to use voice commands on client devices such as handhelds, or any device with a minimal input interface. Applications in either scenario employ Speech Application Language Tags (SALT) to convert voice commands to text-based requests. Microsoft is one of the founders of the SALT Forum (http://www.saltforum.org/) that is promoting this standard. Other speech-enabling standards predate the SALT spec, most notably VoiceXML (http://www.voicexml.org/), which is backed by AT&T, IBM, Nuance, and others. The two standards overlap somewhat, but SALT was designed from its inception to work in either pure telephony apps or in multimodal web apps.

Regardless of which companies are backing these standards, some software developers may be reluctant to devote resources to new speech-enabling initiatives. One of the design goals of Microsoft's SASDK is to make it easier to voice-enable existing apps without requiring major rewrites. Microsoft has also launched the Speech Partners Program, which will provide members with marketing support and guidance in designing speech-based apps.

For now, you can't yet have the company phones answered by Microsoft Sam or other TTS characters—the release date for MSS has not been set. But when it's finally ready, MSS will enter a competitive field of speech-enabling tools from companies including Intervoice, Nuance, ScanSoft, and Envox. This summer Envox released Envox 5 Studio, a platform for developing and deploying speech apps. Envox supports the VoiceXML standard, but it has also joined Microsoft's SPP and has pledged its support for the SALT spec. Despite the competitive environment, most of the speech technology vendors are advocating open APIs and adopting open standards such as Speech Recognition Grammar Specification (SRGS) and Speech Synthesis Markup Language (SSML).

Speech technologies will take center stage at the upcoming SpeechTEK 2003 conference in New York (http://www.speechtek.com/). Microsoft will also be talking about the integration of speech technologies at this year's PDC (http://msdn.microsoft.com/events/pdc/). The lineup of PDC talks include "Building Applications with Advanced Input in 'Longhorn': Inking, Speech, and Beyond" and "Speech-Enabling ASP.NET Applications with the New Microsoft Speech Server." Are you developing speech technologies currently, or are just talking about it? Either way, I'd like to hear what your experiences have been. Send me a note to wdeditor@cmp.com.

John Dorsey
Editor in Chief
wdeditor@cmp.com

MICHAEL J. HUNTER

URL Canonicalization Testing

A test-case generator for decoding URLs

IF YOU HAVE FILLED out web-based forms whose data was passed onto the web server via the URL, you have probably seen sequences of UTF-8 encoded characters such as "%20". The "%nn" encoding is simply the ASCII value of a character converted to hexadecimal and prepended with a percent sign.

If you're writing code that takes a URL as a parameter, the URL may really be a file, and the file's extension may be used to determine what you should do with it (e.g., .asp files are handled differently than .txt files). If the name has been encoded—thus making the filename MyBigFile%2etxt—you won't be able to determine what to do. This typically results in one of two actions: Either an error is returned, or the file will be passed on through the system (after all, it's not an executable, right?). Annoying the user is generally bad, but even worse: If some function down the call chain (or an API the function calls) does decode the filename, you may find yourself running malicious executables.

As you likely know, a web site domain or server name is really just a textual alias for the sequence of four numbers separated by decimal points that make up the IP address (commonly called a "dotted quad"). What you might not know is that the sequence can also be represented in six other forms, and each of these forms can be in decimal, octal, hex, or any combination thereof. These different forms all resolve down to the same machine, so these variations may seem harmless. Once again, if you are deciding what privileges to allow a request based on a dotted-quad IP address and you encounter an IP address without any dots, you might treat it as a domain name. Older versions of Internet Explorer were designed so that if a domain name did not contain any dots, it was assumed to be on an intranet and thus was given higher privileges. This insecure behavior has since been corrected (see References).

These are both examples of a larger problem wherein URLs can be mangled in many different ways—decanonicalized forms, if you will. In this article, I will explain many of the most common ways to munge a URL and present a test case generator that creates each form for a specific URL. After I finish scaring you, I will show how to obviate most of these problems. Converting a URL to its canonical form is actually fairly easy to do. It requires much more than simply converting all "%20" sequences to space characters, however, and most people don't get it right the first time.

User Authentication

One of the simplest ways to mangle a URL is to add user-authentication information. If access to a web page is restricted via a username and password, that login information can be passed as part of the URL: http://username:password@<URL>, where everything between the protocol and the '@' compose the login data. If access is not restricted, login information can still be provided but will be ignored. "Ignored" often translates to "do whatever you like," and such is the case here. So, "http://www.windevnet.com@www.hackme.com" appears to be taking you to the *Windows Developer* web site but, in fact, is loading data from hackme.com. To make matters worse, the '@' can be encoded, giving you "http://www.windevnet.com%40www.hackme.com," making the URL appear even more like the *Windows Developer* web page.

The test case generator creates a few user authentication variations of the base URL by inserting "username:password@", "blahblah@", and "www.evil.com%2fYou%2fAre%2fSo%2fHacked.htm@" between the URL's protocol and the URL's path (the forward slash has to be encoded for this to work when subpaths are included). It also partially and fully encodes the '@' to the requested level using the specified encoder; I'll cover encoders and encoding levels later.

Navigation Injection

The next simplest—conceptually, at least—method for mangling URLs is navigation injection. The '.' and '..' characters are typically used to pull in images and other supporting files (e.g.,), and this pathname navigation works just as well in URLs: http://www.evil.com/../../../webserver/data/passwords.txt. If your web server doesn't guard against this, a hacker can easily gain access to private data or run commands on your web server: http://www.evil.com/../../../windows/system32/cmd.exe /c format c:. Similarly, if you are using the URL path to determine access privileges, an attacker (or simply an employee curious what the CEO's salary is) can gain access to data she should not have. While the need

MICHAEL J. HUNTER *is a lead developer at Humbug Reality where he works on whatever catches his interest. Michael's alter ego is a tester at a major software company who has so much fun finding bugs he doesn't mind funding Michael's flights of fancy.*

to conjure up the correct relative path to access specific folders may seem to lessen this threat, in reality, many administrators install web servers to their default location. Attackers can install web servers just as easily as we can, so it is not very hard for them to determine the exact path to use.

An infinite number of path injection cases are possible, but one of the goals of the test-case generator is to only generate valid cases. Thus, it restricts itself to inserting "this folder" navigation as well as "parent folder" navigation when the URL path contains multiple levels. `CNavigationInjector` (Listing 1) takes a path and passes it through a `CUrl` (a helper object that knows how to split a path or URL into its protocol, domain, and path) to isolate the path, then uses `SelectInjectionPoint` to decide where to insert the path navigation.

`SelectInjectionPoint` first identifies the location of each path separator, then picks one as follows:

- If three or more separators were found, have `SelectRandomIndex` (a thin wrapper around Boost's random number generator; see References) pick any after the first. (I explicitly ignore the first index to ensure valid parent path navigation.)

Listing 1 CNavigationInjector

```
// from header:
// std::vector<std::wstring> paths;
// const std::wstring pathSeparatorCharacters;

CNavigationInjector::CNavigationInjector(std::wstring path)
    : pathSeparatorCharacters(L"/\\")
    {
    // We should have been given just the path, but we'll run it through a Url
    //    just to be sure.
    CUrl url(path);
    if (0 < url.Path().length())
        {
        unsigned long injectionPoint(SelectInjectionPoint(url.Path()));
        InjectNavigationAtPoint(url.Path(), injectionPoint);
        }
    }
void CNavigationInjector::InjectNavigationAtPoint(std::wstring path
    , unsigned long injectionPoint)
    {
    InjectForwardSlashDot(path, injectionPoint);
    InjectForwardSlashDotDot(path, injectionPoint);
    }
void CNavigationInjector::InjectForwardSlashDot(std::wstring path
    , unsigned long injectionPoint)
    {
    paths.push_back(path.insert(injectionPoint + 1, L"./"));
    }
void CNavigationInjector::InjectForwardSlashDotDot(std::wstring path
    , unsigned long injectionPoint)
    {
    std::wstring parentFolder(GetParentFolder(path, injectionPoint));
    if (0 < parentFolder.length())
        {
        paths.push_back(path.insert(injectionPoint + 1, L"../"
          + parentFolder));
        }
    }
unsigned long CNavigationInjector::SelectInjectionPoint(std::wstring path) const
    {
    // Count the number of path separator characters in the path.
    InjectionPointList possibilities;
    std::wstring::size_type location(0);
    while (std::wstring::npos !=
        (location = path.find_first_of(pathSeparatorCharacters, location)))
        {
        possibilities.push_back(location);
        ++location;
        }

    // Pick one of the locations as the injection point.  We only pick a
    // random point if there are more than two path separators found (and
    // don't consider the first location as a possibility) to ensure we can
    // construct a valid path when ".." is injected.
    if (2 < possibilities.size())
        {
        unsigned long index(SelectRandomIndex(2, possibilities.size()));
        unsigned long position(possibilities[index - 1]);
        return position;
        }
    else if (2 == possibilities.size())
        {
        //------------------------------------------------
        // If we found two path separators the first one should be the root
        // and the second one should be trailing the top-level folder; it's
        // this end location we want to modify.
        return possibilities[1];
        }
    else
        {
        //------------------------------------------------
        // There are either zero or one path separators found; the first should
        // be the root, so we return position zero either way.
        return 0;
        }
    }

std::wstring CNavigationInjector::GetParentFolder(std::wstring path
    , unsigned long childPathStartPoint) const
    {
    std::wstring::size_type parentPathStartPoint(path.find_last_of(
        pathSeparatorCharacters, (childPathStartPoint - 1)));
    if (std::wstring::npos != parentPathStartPoint)
        {
        return path.substr(parentPathStartPoint + 1
            , (childPathStartPoint - parentPathStartPoint));
        }
    else
        {
        return L"";
        }
    }
```

Listing 2 String-to-number conversion

```
std::wstring CConverters::NumberToStringAsBase(unsigned long value
    , unsigned long base, unsigned long minimumWidth, wchar_t fillCharacter)
    {
    std::wstring baseNValue;
    try
        {
        std::wstringstream interpreter;
        if (!(interpreter << std::setbase(base))
            || !(interpreter << std::setw(minimumWidth))
            || !(interpreter << std::setfill(fillCharacter))
            || !(interpreter << value)
            || !(interpreter >> baseNValue)
            || !(interpreter >> std::ws).eof())
            {
            return L"";
            }
        }
    catch (...)
        {
        return L"";
        }
    return baseNValue;
    }
unsigned long CConverters::StringAsBaseToNumber(std::wstring value
        , unsigned long base)
    {
    if (0 == value.length())
        {
        return 0;
        }

    unsigned long numericValue;
    try
        {
        std::wstringstream interpreter;
        if (!(interpreter << std::setbase(base))
            || !(interpreter << value)
            || !(interpreter >> numericValue)
            || !(interpreter >> std::ws).eof())
            {
            return 0;
            }
        }
    catch (...)
        {
        return 0;
        }
    return numericValue;
    }
```

- If exactly two separators are found, pick the second (again to ensure valid parent path navigation).
- Otherwise, pick the zero'th index, causing the navigation to be injected at the root. (No parent path navigation will be injected.)

Once we have the injection point, it's a simple matter for `InjectForwardSlashDot` to insert the "this folder" navigation. `InjectForwardSlashDotDot` has it slightly more complicated as it can only do so if a parent folder exists before the injection point. `GetParentFolder` determines whether this is so by searching the path backwards from the injection point for another path separator; if one is found, the two locations define the parent folder, which is used to generate the "..\parent_folder" string that is inserted.

If you download and peruse the full code, you'll note that both forward and backward slash-path navigation variants are generated. Web browsers generally support both, so both need to be tested.

IP Address Encoding

Now we start getting into the truly interesting problems. Although they tend to require complicated mathematics, they also tend to be eminently computable, so we can let the test-case generator do all the nasty calculations, and we can concentrate on correctly handling the cases. I touched upon the first of these earlier. The most common form of an IP address is the familiar dotted decimal quad: www.windevnet.com translates to 66.35.216.85, for example. As the name implies, these numbers are in base 10. As is often the case when working with computers, though, the numbers can just as easily be given in base 8 (octal) or base 16 (hexadecimal, or hex). All that's required is prepending a '0' for octal or a '0x' for hex.

C++ makes converting between strings and numbers or converting a number between bases super easy. Decisions you've chewed over in the past—such as trying to guess the largest number you'll need to handle so you can size the buffer big enough to handle all possible values but not so big you're just wasting space—are completely eliminated when you take advantage of the `stringstream` class. In most cases, you can simply use Boost's `lexical_cast`, but as it doesn't provide a way to specify the base of a number, I borrowed its technique to roll my own converter. `NumberToStringAsBase` (see Listing 2) uses the `std::stringstream` object—a member of the stream I/O family that happens to use a string as its backing store—to read in a number and extract its string form. Not only does `stringstream` take care of the datatype conversion, but it also allows you to specify which base to use. `StringAsBaseToNumber` does the reverse: It converts a stringized number in a given base to the equivalent number.

Not only can the quads be encoded in different bases, but additional digits can be prepended to the octal and hex forms. Each quad is really an 8-bit value, but IP-processing functions often accept larger numbers and simply ignore the extra bits. To use *Windows Developer*'s address again, 66.35.216.85 could also be written as 66.35.216.3157. Thus, not only do we need to generate variations where the various parts of the IP address are encoded as hex and octal, we need to generate variations containing random data prepended to each quad. Given our string-to-number converters and the `SelectRandomIndex` helper, this becomes as easy as generating a series of random numbers; see Listing 3.

Finally, two or more of the quads can be collapsed to a single value. An IP address is really a single number split into pieces to make it easier

Listing 3 Generating a series of random numbers

```
std::wstring CIPEncoder::GenerateRandomDecimalDigits(
    unsigned long maximumNumberOfDigits) const
    {
    return L"";
    }
std::wstring CIPEncoder::GenerateRandomHexDigits(
    unsigned long maximumNumberOfDigits) const
    {
    std::wstring returnValue(L"");

    if (0 < maximumNumberOfDigits)
        {
        unsigned long digitsToGenerate(
            SelectRandomIndex(0, maximumNumberOfDigits));
        for (unsigned long currentDigit = 0; currentDigit < digitsToGenerate;
            ++currentDigit)
            {
            returnValue += CConverters::NumberToStringAsBase(
                SelectRandomIndex(0, 15), 16);
            returnValue += CConverters::NumberToStringAsBase(
                SelectRandomIndex(0, 15), 16);
            }
        }
    return returnValue;
    }
std::wstring CIPEncoder::GenerateRandomOctalDigits(
    unsigned long maximumNumberOfDigits) const
    {
    std::wstring returnValue(L"");

    if (0 < maximumNumberOfDigits)
        {
        unsigned long digitsToGenerate(
            SelectRandomIndex(0, maximumNumberOfDigits));
        for (unsigned long currentDigit = 0; currentDigit < digitsToGenerate;
            ++currentDigit)
            {
            returnValue += CConverters::NumberToStringAsBase(
                SelectRandomIndex(0, 7), 8);
            }
        }

    return returnValue;
    }
```

Listing 4 Collapsing IP quads

```
std::wstring CIPEncoder::CollapseQuads(std::wstring quadOne
    , std::wstring quadTwo) const
    {
    return CollapseQuads(quadOne, quadTwo, L"", L"");
    }
std::wstring CIPEncoder::CollapseQuads(std::wstring quadOne
    , std::wstring quadTwo, std::wstring quadThree) const
    {
    return CollapseQuads(quadOne, quadTwo, quadThree, L"");
    }
std::wstring CIPEncoder::CollapseQuads(std::wstring quadOne
    , std::wstring quadTwo, std::wstring quadThree, std::wstring quadFour) const
    {
    // An empty string converts to a single 0, but to make the math work
    // correctly we need two-digit hex values.  Thus we prepend the missing
    // 0 if the conversion gives an odd number of digits.
    quadOne = CConverters::DecimalToHex(quadOne);
    if (1 == (quadOne.length() % 2))
        {
        quadOne = L"0" + quadOne;
        }
    quadTwo = CConverters::DecimalToHex(quadTwo);
    if (1 == (quadTwo.length() % 2))
        {
        quadTwo = L"0" + quadTwo;
        }
    quadThree = CConverters::DecimalToHex(quadThree);
    if (1 == (quadThree.length() % 2))
        {
        quadThree = L"0" + quadThree;
        }
    quadFour = CConverters::DecimalToHex(quadFour);
    if (1 == (quadFour.length() % 2))
        {
        quadFour = L"0" + quadFour;
        }
    return CConverters::HexToDecimal(quadOne + quadTwo + quadThree + quadFour);
    }
```

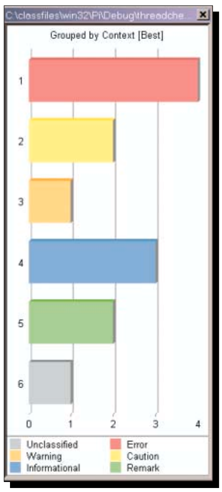
Introducing a Powerful New Debugging Tool for Threaded Applications

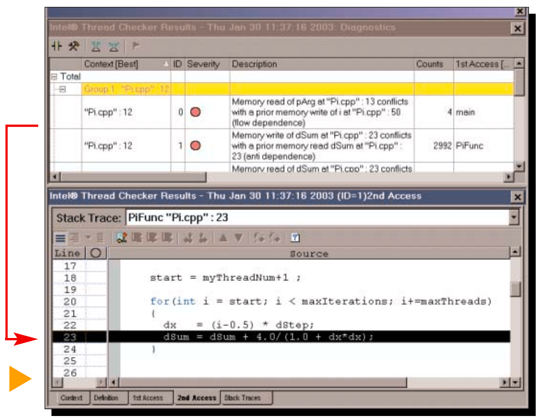
Intel® Thread Checker 1.0 for Windows*

Intel® Thread Checker 1.0 for Windows* helps you debug Win32* and OpenMP* threaded applications. The Intel® Thread Checker uses an advanced error detection engine to find bugs that might otherwise go undetected in your QA process. Quickly find threading bugs that would take days or weeks to find using traditional tools and methods!

If you work with threaded code, you should download a free trial. When you purchase Intel® Thread Checker you also receive the Intel® VTune™ Performance Analyzer.

Error Classification *Identifies 6 levels of threading issues from errors and warnings to informative comments*





Diagnostics View *Lists specific information on each error— race conditions, stalled threads, deadlocks and more…*

Source Code View *Clicking in diagnostics view takes you directly to the specific source code line.*

Intel® VTune™ Performance Analyzer 7.0

The award-winning VTune™ Performance Analyzer helps you improve your application performance by enabling you to locate and remove bottlenecks in your code. Features like the Intel Tuning Assistant give detailed guidance on tuning your code.

"Using Intel Thread Checker we discovered two elusive bugs on the very first day..."

—Farzin Shakib
President ACUSIM Software, Inc.

YOU SAVE UP TO $272!	Paradise #	Retail	Discount
Intel® Thread Checker 1.0 for Windows (includes VTune™ Performance Analyzer)	I23 0A3A	$1,198.00	**$925.99**
Intel® VTune™ Performance Analyzer v7.0	I23 0A2N	$699.00	**$599.99**

Download a Free Trial at:
programmersparadise.com/intel/wdj

Programmer's Paradise®
To order or request additional information call:
800-423-9990
Email: intel@programmers.com

03CT010F

to work with. Four small numbers are easier both on a human level (people usually find many small values easier to work with than a single large value—would you rather read off 66.35.216.85 or 1109645397 when recording a user's IP address?) and on a technical level (for example, a subnet mask of 0.0.255.0 lets you easily separate every IP address from 66.35.0.0 to 66.35.255.255).

Collapsing two, three, or all four of the quads is valid (when all four are collapsed, the IP address is called "dotless"). Many web sites will give you a somewhat involved formula for

Listing 5 A stripped-down version of GenerateEncodings

```
// from header:
// typedef std::wstring QuadS;
// const std::wstring quadSeparator(L".");
// const std::wstring hexPrefix(L"0x");
// const std::wstring octalPrefix(L"0");

void CIPEncoder::GenerateEncodings(QuadS quadOne, QuadS quadTwo
    , QuadS quadThree, QuadS quadFour
    , unsigned long charactersToPrepend, std::wstring dotEncoding)
    {
    // Raw IP.
    AddAddress(GenerateRandomDecimalDigits(charactersToPrepend) + quadOne
        + quadSeparator
        + GenerateRandomDecimalDigits(charactersToPrepend) + quadTwo
        + quadSeparator
        + GenerateRandomDecimalDigits(charactersToPrepend) + quadThree
        + quadSeparator
        + GenerateRandomDecimalDigits(charactersToPrepend) + quadFour);

    // Encode as hex.
    AddAddress(hexPrefix + GenerateRandomHexDigits(charactersToPrepend)
            + CConverters::DecimalToHex(quadOne)
        + quadSeparator
        + GenerateRandomDecimalDigits(charactersToPrepend) + quadTwo
        + quadSeparator
        + GenerateRandomDecimalDigits(charactersToPrepend) + quadThree
        + quadSeparator
        + GenerateRandomDecimalDigits(charactersToPrepend) + quadFour);

    // Collapse the quads.
    AddAddress(GenerateRandomDecimalDigits(charactersToPrepend)
            + CollapseQuads(quadOne, quadTwo)
        + quadSeparator
        + GenerateRandomDecimalDigits(charactersToPrepend) + quadThree
        + quadSeparator
        + GenerateRandomDecimalDigits(charactersToPrepend) + quadFour);
    AddAddress(GenerateRandomDecimalDigits(charactersToPrepend)
        + CollapseQuads(quadOne, quadTwo, quadThree, quadFour));

    // Collapse the quads and encode as hex.
    AddAddress(hexPrefix + GenerateRandomHexDigits(charactersToPrepend)
            + CConverters::DecimalToHex(CollapseQuads(quadOne, quadTwo))
        + quadSeparator
        + GenerateRandomDecimalDigits(charactersToPrepend) + quadThree
        + quadSeparator
        + GenerateRandomDecimalDigits(charactersToPrepend) + quadFour);

    // Encode the dots.
    AddAddress(GenerateRandomDecimalDigits(charactersToPrepend) + quadOne
        + dotEncoding
        + GenerateRandomDecimalDigits(charactersToPrepend) + quadTwo
        + quadSeparator
        + GenerateRandomDecimalDigits(charactersToPrepend) + quadThree
        + quadSeparator
        + GenerateRandomDecimalDigits(charactersToPrepend) + quadFour);

    // QUAD_CONVERTED  QUAD_SYSTEM  QUAD_COLLAPSED  EXTRA_CHARS  DOTS_ENCODED
    // None            Decimal      None            None         None
    AddAddress(quadOne + quadSeparator + quadTwo + quadSeparator + quadThree
        + quadSeparator + quadFour);
    //    None         Decimal      None            P2           Second
    AddAddress(quadOne + quadSeparator
        + GenerateRandomDecimalDigits(charactersToPrepend) + quadTwo
        + dotEncoding + quadThree + quadSeparator + quadFour);
    //    Some         Decimal      P3+P4           P1           First
    AddAddress(hexPrefix + GenerateRandomHexDigits(charactersToPrepend)
            + CConverters::DecimalToHex(quadOne)
        + dotEncoding + quadTwo
        + octalPrefix + GenerateRandomOctalDigits(charactersToPrepend)
            + CConverters::DecimalToOctal(CollapseQuads(quadThree, quadFour)));
    //    All          Octal        All             None         Third
    AddAddress(octalPrefix + CConverters::DecimalToOctal(
        CollapseQuads(quadOne, quadTwo, quadThree, quadFour)));
    }
```

Listing 6 ConvertToUtf8

```
// from the header:
// const unsigned long utfPrimaryLeadByte_0;
// const unsigned long utfPrimaryLeadByte_1110;
// const unsigned long utfPrimaryLeadByte_11110;
// const unsigned long utfSecondaryLeadByte;
// unsigned long overlongUtfExtraBytesToAdd; <== constructor parameter

std::wstring CUtf8Encoder::ConvertToUtf8(wchar_t characterToEncode) const
    {
    typedef const unsigned long Byte;
    const std::wstring prefix(L"%");

    std::vector<std::wstring> encodings;

    const unsigned long codepoint(
        static_cast<const unsigned long>(characterToEncode));
    if (codepoint <= 0x7F)
        {
        Byte byteOne(utfPrimaryLeadByte_0 | codepoint);
        encodings.push_back
            (
            prefix + CConverters::NumberToStringAsBase(byteOne, 16, 2)
            );
        }
    if (codepoint <= 0x7FF)
        {
        Byte byteOne(utfSecondaryLeadByte | GetRightBits(codepoint, 1, 6));
        Byte byteTwo(utfPrimaryLeadByte_1110 | GetRightBits(codepoint, 7, 5));
        encodings.push_back
            (
              prefix + CConverters::NumberToStringAsBase(byteTwo, 16, 2)
            + prefix + CConverters::NumberToStringAsBase(byteOne, 16, 2)
            );
        }
    if (codepoint <= 0xFFFF)
        {
        Byte byteOne(utfSecondaryLeadByte | GetRightBits(codepoint, 1, 6));
        Byte byteTwo(utfSecondaryLeadByte | GetRightBits(codepoint, 7, 6));
        Byte byteThree(utfPrimaryLeadByte_11110
            | GetRightBits(codepoint, 13, 4));
        encodings.push_back
            (
              prefix + CConverters::NumberToStringAsBase(byteThree, 16, 2)
            + prefix + CConverters::NumberToStringAsBase(byteTwo, 16, 2)
            + prefix + CConverters::NumberToStringAsBase(byteOne, 16, 2)
            );
        }
    // And so on and so forth through the rest of the character ranges.
    }

    if (overlongUtfExtraBytesToAdd < encodings.size())
        {
        return encodings[overlongUtfExtraBytesToAdd];
        }
    else
        {
        return encodings[encodings.size() - 1];
        }
    }
unsigned long CUtf8Encoder::GetRightBits(unsigned long value
    , unsigned long indexOfFirstBit, unsigned long numberOfBitsToGet) const
    {
    // Pull out the specified number of bits, starting the specified number of
    // of bits from the right-hand side.  For example, given
    // "0110 1011 0001 1101", GetRightBits(3, 6) will return "0000 0111".
    return RotateBitsRight(value, indexOfFirstBit - 1)
        & GetRightAlignedMaskBits(numberOfBitsToGet);
    }
unsigned long CUtf8Encoder::GetRightAlignedMaskBits(
    unsigned long numberOfBitsToMask) const
    {
    // Generate a solid right-aligned bit mask of the specified size.
    return RotateBitsRight(0xFFFFFFFF, (32 - numberOfBitsToMask));
    }
unsigned long CUtf8Encoder::RotateBitsRight(unsigned long value
    , unsigned long numberOfBitsToRotate) const
    {
    // Rotates all bits in the specified value the specified number of digits
    // right, dropping the rightmost bits.
    return value >> numberOfBitsToRotate;
    }
```

Programmer's Paradise®

Your best source for software development tools!

GUARANTEED BEST PRICES*

Should you see one of these products listed at a lower price in another ad in this magazine, CALL US! **We'll beat the price**, and still offer our same quality service and support!

*Terms of the offer:

- Offer good through Oct. 31, 2003
- Applicable to pricing on current versions of software listed
- Oct. issue prices only
- Offer does not apply towards obvious errors in competitors' ads
- Subject to same terms and conditions

PROGRAMMER'S GUARANTEED BEST PRICES PARADISE

Prices subject to change. Not responsible for typographical errors.

XMLSPY 2004 — NEW!

by Altova

XMLSPY 2004 is the industry standard XML Development Environment for designing, editing and debugging enterprise-class applications. Now including:

- Visual Studio® .NET Integration
- XML-to-XML Mapping
- XML Differencing

XMLSPY 2004 is the ultimate productivity enhancer for J2EE, .NET and database developers.

Enterprise Edition
Paradise #
I0D 017T
$965.99

Professional Edition
Paradise #
I0D 017U
$389.99

www.programmersparadise.com/altova

RoboHelp Office

The Industry Standard in Help Authoring

Create professional Help systems for desktop and Web-based applications, including .NET.

- Create all popular Help formats
- Create standard and advanced Help-specific features
- Work in WYSIWYG or true code
- Easily create context-sensitive Help
- Generate printed documentation
- Winner of 55 industry awards

Programmer's Paradise #1 Best-Selling Help Authoring Tool for 7 Years Running!

Paradise #
E75 0311
$859.99*

** Price after mfr's mail-in rebate. New US/Can licenses only. Expires 10/31/03.*

www.programmersparadise.com/ehelp

Paradise Picks

DevTrack 5.5

Powerful Defect and Project Tracking

by TechExcel

DevTrack, the market-leading defect and project tracking solution, comprehensively manages and automates your software development processes. DevTrack 5.5 features sophisticated workflow and process automation, seamless source code control integration with VSS, Perforce and ClearCase, robust searching, and built-in reports and analysis. Intuitive administration and integration reduces the cost of deployment and maintenance.

programmersparadise.com/techexcel

Paradise #
T34 0199
$482.99

LEADTOOLS Document Imaging

by LEAD Technologies

Document imaging including annotations, specialized bitonal (b/w) image display and processing like scale-to-gray and favor-black, performance and memory optimizations for bitonal images, image clean-up like hole-punch, line and staple removal, high-speed scanning.

Paradise #
L05 048V
$1,639.99

www.programmersparadise.com/lead

Intel® C++ and Fortran Compilers

by Intel

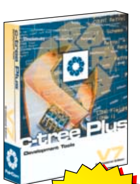
Increase the Performance of Your Application with Intel's High-Performance Compilers — Version 7.0

Intel's expertise in processors shows in this latest release of its flagship compiler product-line: Version 7 of its C++ and Fortran Compilers for Windows and Linux. Intel Compilers deliver outstanding performance on Pentium® 4 and Intel® Xeon® processors, and the 64-bit Itanium and Itanium 2 processors and take advantage of Multi-processor systems and Hyper-Threading technology.

Intel C++ for Windows
Paradise #
I23 0A10
$308.99

www.programmersparadise.com/intel

c-tree Plus®



by FairCom

With unparalleled performance and sophistication, c-tree Plus gives developers absolute control over their data management needs. Commercial developers use c-tree Plus for a wide variety of embedded, vertical market, and enterprise-wide database applications. Use any one or a combination of our flexible APIs including low-level and ISAM C APIs, simplified C and C++ database APIs, SQL, ODBC, or JDBC. c-tree Plus can be used to develop single-user and multi-user non-server applications or client-side application for FairCom's robust database server—the c-tree Server. Windows to Mac to Unix all in one package.

NEW! SQL Support!

Paradise #
F01 0131
$850.99

www.programmersparadise.com/faircom

TX Text Control ActiveX 10.0 — NEW .NET VERSION!

by The Imaging Source

Add RTF, DOC, HTML, CSS and PDF Support to Your Application

TX Text Control is royalty-free, robust and powerful word processing software in reusable component form. The new Enterprise/XML version features a rich set of properties for the manipulation of XML and CSS. Developers can now offer end-users the ability to separate their textual content from their formatting rules.

Download a demo today. DEMO AVAILABLE programmersparadise.com

Enterprise Edition
Paradise #
T79 0214
$1,495.99

Professional Edition
Paradise #
T79 0215
$729.99

www.programmersparadise.com/theimagingsource

Sun™ ONE Studio 5, Standard Edition

by Sun Microsystems

Sun ONE Studio 5, Standard Edition is an integrated development environment (IDE) for Java technology. Based on the modular, open source NetBeans Tools Platform, this IDE is ideal for building and deploying Web services across multiple hardware and software platforms, including Windows, Windows NT, Linux, and the Solaris Operating Environment.

www.programmersparadise.com/sunone

Paradise #
S69 0U67
$556.99

SourceOffSite Classic — New Version! 3.5.3

by SourceGear Corp.

Access SourceSafe™ over the Internet

Visual SourceSafe collaboration tool for distributed teams.

- IDE integration
- Over 10 times faster than RAS
- Efficient—uses data compression
- Secure—up to 128-bit data encryption
- Familiar—the look and feel of VSS
- Cross-Platform—Windows and UNIX.

† Classic edition plus Media, 1 user.

Paradise #
S0Z 041C
$223.99†

www.programmersparadise.com/sourcegear

PR-Tracker™ v5.1

by Softwise Company

Affordable scalable enterprise level bug tracking system featuring classification, assignment, sorting, searching, reporting, access control, user permissions, attachments and email notification. Integrates with PR-Tracker Web Client (included) and ProblemReport.asp (included for your betatest or customer support interface). Supports Access and SQL Server.

Download Today! DOWNLOAD NOW! programmersparadise.com

Paradise #
S3R 0147
$117.99

www.programmersparadise.com/softwise

800-445-7899 • programmersparadise.com

determining the collapsed value. "How to Obscure Any URL" (http://www.pc-help.org/obscure.htm) explains it particularly well, but I find it much simpler to simply convert each quad to its hexadecimal form, squish the four hex values together, then convert the resulting hex value to the required base. Thus 66.35.216.85 becomes 42.23.D8.55, which becomes 4223D855, which becomes the 1109643597 I previously referenced.

Once again, our string-to-number converters make short work of this task; see Listing 4. The closest to complicated this comes is remembering to prepend a 0 to single-digit values to make the math work correctly.

The `CIPEncoder` class brings all this together. The constructor does the grunt work of converting the domain name to an IP address and splitting the address into its constituent quads, then it has `GenerateEncodings` (Listing 5 is a stripped-down version) generate a set of encodings using the provided encoder. `GenerateEncodings` converts each quad to hex and octal, then it generates each collapsed combination and also converts them to hex and octal. Finally a series of mix-and-match cases are generated where one or more quads are converted to a different base while one or more (possibly different) quads are collapsed; I used multiwise combinatorics to minimize the number of these cases while still ensuring fairly complete coverage across the myriad of possibilities. And, of course, every case has random digits prepended to one or more of the quads.

UTF-8 Character Encodings

Now we're finally ready to talk about character encodings. In the introduction, I mentioned the UTF-8 encoding; this is the most common but UTF-16, UTF-32, and Unicode encodings also exist. Almost any character in a URL can be encoded: the slashes in the protocol, the dots in the IP address, the dots or any other character in the domain name, and any character in the path (including path separators). Further, encodings can themselves be encoded, so "file.txt" can become "file%2Etxt", which can become "file%252Etxt" or "file%25%32%65txt"; a further pass could take it on to "file%25252Etxt" or "file%25%32%6525-%32%65txt". And, of course, any character can be encoded in each pass—not just a character that was encoded in the previous pass—so a single URL could contain characters that aren't encoded at all, characters that have been encoded once or twice, and characters that have been encoded 10 or more times.

As long as you're dealing with ASCII values, UTF-8 encoding is simple: Simply prepend a '%' to the hexadecimal numerical value of the character. Thus, a period, which has an ASCII value of 46, or 0x2E, is encoded as "%2E". Once you expand out into the wide world of Unicode, however, the math starts getting a little hairy. UTF-8 is a variable-width encoding that specifies the number of subsequent bytes that go with the lead byte by using special sequences of lead bits in the lead byte, a special "I'm a supporting byte" bit sequence in subsequent bytes, and spreading the bits composing the character's value across the remaining bits in each byte.

Even with examples and assistance from a colleague, Lawrence Landauer, it took me awhile to understand how this works. Let's start simple: A period is 0x2E, which is 101110 in binary. Looking through Table 1 for the character range that contains our character, we see that it falls into the very first range, so we replace the 'x's in the Encoded Bytes cell with the binary representation of our character. This gives us 00101110, which converts back to hex as 0x2E. We already knew that would be the answer, but it gives us confidence our technique is correct. Now we can follow the exact same process for any other value: convert the value to binary, find the character range containing the value, replace the 'x's in the Encoded Bytes cell with the binary value, then convert back to hex. Thus, we can convert the Lira sign to UTF-8 like so:

1. Look up the character's value. (If you have access to Windows, charmap.exe is very helpful in this regard. If you have Microsoft Office, you'll also want to install the Arial Unicode MS font by checking the Office Shared Features | International Support | Universal Font option.) The Lira sign is 0x20A4.
2. Convert the value to binary. (Again, if you have access to Windows, calc.exe helps out here.) The Lira sign in binary is 10000010100100.
3. Look through Table 1 and find the character range containing the value, then move across and find the byte encoding. 0x20A4 falls between 0x800 and 0xFFFF, so we use 1110xxxx 10xxxxxx 10xxxxxx.
4. Moving right-to-left both in the byte-encoding template and in the binary value,

Table 1 UTF-8 Character Mappings

Character Range	Encoded Bytes
0x00000000ñ0x0000007F	0xxxxxxx
0x00000080ñ0x000007FF	110xxxxx 10xxxxxx
0x00000800ñ0x0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
0x00010000ñ0x001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
0x00200000ñ0x03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
0x04000000ñ0x7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx, 10xxxxxx

Listing 7 CStringEncoder header

```
#ifndef CStringEncoder_h
#define CStringEncoder_h

#include <assert.h>
#include <string>
#include <vector>
#include "IEncoder.h"

typedef std::vector<std::wstring> EncodingsBag;

class CStringEncoder
    {
    private:
    EncodingsBag encodings;

    public:
    CStringEncoder(std::wstring stringToEncode, unsigned long level
        , IEncoder & encoder);
    public:
    virtual ~CStringEncoder();

    public:
    //-----------------------------------------------
    // Valid indices are zero through Count - 1.
    unsigned long Count() const;
    std::wstring Item(unsigned long index) const;

    private:
    void AddEncoding(std::wstring encodingToAdd);
    bool DontAlreadyHaveEncoding(std::wstring encodingToAdd) const;

    void CreateEntireStringFullEncodings(std::wstring stringToEncode
        , unsigned long level, IEncoder & encoder);
    void CreateEntireStringRandomEncodings(std::wstring stringToEncode
        , unsigned long level, IEncoder & encoder);
    void CreateSingleCharacterEncodings(std::wstring stringToEncode
        , unsigned long level, IEncoder & encoder);
    };
#endif // CStringEncoder_h
```

replace each 'x' in the template with the corresponding bit from the binary value. Replace any remaining 'x's with 0. This gives us 11100010 10000010 10100100.

5. Convert the binary value to hex. This gives us 0xE2 0x82 0xA4.

If my explanation isn't helping, the book *Writing Secure Code* has a readable explanation of all this (see References); or if you're a glutton for punishment, RFC 2279 (http://www.ietf.org/rfc/rfc2279.txt) is the official definition.

We programmers are never one to leave a good thing alone. While the UTF-8 makes it quite clear you should always use the shortest possible representation to encode a character, you are free to use a longer (known as "overlong") encoding as well. If we repeat the Lira example, but use the immediately following template (11110xxx 10xxxxxx 10xxxxxx 10xxxxxx), we'll end up with a result of 0xF0 0x82 0x82 0xA4.

Lest you think you won't run into this in the wild, Microsoft Security Bulletin MS00-057 describes a bug in Microsoft Internet Information Server exploitable via overlong UTF-8 (which the bulletin coyly calls "a particular type of malformed URL") that allowed an attacker to bypass security and use path navigation injection to pass commands to the command shell.

As it happens, the test-case generator will create UTF-8 and overlong UTF-8 cases for you via the `CUtf8Encoder` class. `ConvertToUtf8` (see

Listing 8 CStringEncoder body

```
#include "stdafx.h"
#include "StringEncoder.h"

#include <algorithm>

#include "CharacterEncoder.h"

CStringEncoder::CStringEncoder(std::wstring stringToEncode, unsigned long level
    , IEncoder & encoder)
    {
    CreateSingleCharacterEncodings(stringToEncode, level, encoder);
    CreateEntireStringFullEncodings(stringToEncode, level, encoder);
    CreateEntireStringRandomEncodings(stringToEncode, level, encoder);
    }
CStringEncoder::~CStringEncoder()
    {
    }

unsigned long CStringEncoder::Count() const
    {
    return encodings.size();
    }
std::wstring CStringEncoder::Item(unsigned long index) const
    {
    return encodings[index - 1];
    }

void CStringEncoder::AddEncoding(std::wstring encodingToAdd)
    {
    if (DontAlreadyHaveEncoding(encodingToAdd))
        {
        encodings.push_back(encodingToAdd);
        }
    }
bool CStringEncoder::DontAlreadyHaveEncoding(std::wstring encodingToAdd) const
    {
    return (encodings.end() == std::find(encodings.begin(), encodings.end()
        , encodingToAdd));
    }

void CStringEncoder::CreateEntireStringFullEncodings(
    std::wstring stringToEncode, unsigned long level, IEncoder & encoder)
    {
    // We shouldn't ever get an empty string, but if we do we just won't
    //    encode it.
    if (0 == stringToEncode.length())
        {
        AddEncoding(stringToEncode);
        return;
        }

    for (unsigned long currentLevel = 0; currentLevel <= level; ++currentLevel)
        {
        std::wstring encoding(L"");

        // Replace every character in the string with its full encoding.
        for (std::wstring::size_type currentCharacter = 0;
            currentCharacter < stringToEncode.length(); ++currentCharacter)
            {
            CCharacterEncoder characterEncoder(stringToEncode[currentCharacter]
                , currentLevel, encoder);
            encoding += characterEncoder.FullEncoding(currentLevel);
            }

        AddEncoding(encoding);
        }
    }

void CStringEncoder::CreateEntireStringRandomEncodings(
    std::wstring stringToEncode, unsigned long level, IEncoder & encoder)
    {
    // We shouldn't ever get an empty string, but if we do we just won't
    //    encode it.
    if (0 == stringToEncode.length())
        {
        AddEncoding(stringToEncode);
        return;
        }

    for (unsigned long currentLevel = 0; currentLevel <= level; ++currentLevel)
        {
        std::wstring encoding(L"");

        // Replace every character in the string with either its
        //    full or partial encoding.
        for (std::wstring::size_type currentCharacter = 0;
            currentCharacter < stringToEncode.length(); ++currentCharacter)
            {
            CCharacterEncoder characterEncoder(stringToEncode[currentCharacter]
                , currentLevel, encoder);
            // Decide which character encoding level to use.  If we're on
            //   level 0 we can skip this step.
            unsigned long levelToUse(0);
            if (0 < currentLevel)
                {
                levelToUse = SelectRandomIndex(0, currentLevel);
                }
            if (UseFullEncoding())
                {
                encoding += characterEncoder.FullEncoding(levelToUse);
                }
            else
                {
                encoding += characterEncoder.PartialEncoding(levelToUse);
                }
            }

        AddEncoding(encoding);
        }
    }
void CStringEncoder::CreateSingleCharacterEncodings(std::wstring stringToEncode
    , unsigned long level, IEncoder & encoder)
    {
    // We shouldn't ever get a null string, but if we do we just won't
    //    encode it.
    if (0 == stringToEncode.length())
        {
        AddEncoding(stringToEncode);
        return;
        }

    // Decide which character to replace.  If the string is only one
    //    character long we can skip this step.
    unsigned long indexToReplace(0);
    if (1 < stringToEncode.length())
        {
        indexToReplace = SelectRandomIndex(0, stringToEncode.length() - 1);
        }
    CCharacterEncoder characterEncoder(stringToEncode[indexToReplace], level
        , encoder);

    // Add a version of the base string with the chosen character replaced
    //    with its partial and full encoding at each level.
    for (unsigned long currentLevel = 0; currentLevel <= level; ++currentLevel)
        {
        std::wstring partialEncoding(stringToEncode);
        partialEncoding.replace(indexToReplace, 1
            , characterEncoder.PartialEncoding(currentLevel));
        AddEncoding(partialEncoding);

        std::wstring fullEncoding(stringToEncode);
        fullEncoding.replace(indexToReplace, 1
            , characterEncoder.FullEncoding(currentLevel));
        AddEncoding(fullEncoding);
        }
    }
```

Listing 6) goes through the same steps we did to convert a character's value to UTF-8. After finding the first character range into which the character fits, it spreads the character's value's bits across the empty bits of the template, then converts each byte to its equivalent string-ized hex value. It doesn't stop there, however, but goes on to create all valid overlong encodings as well. `CUtf8Encoder`'s constructor allows you to specify whether to generate overlong UTF-8 and how overlong to go; `ConvertToUtf8` uses this to determine which of the generated encodings to return.

Listing 9 Generating and using the test cases

```
#import "CommonTestCases.dll" rename_namespace("TestCases")

void UseEncodings()
    {
    TestCases::IFactoryPtr factory;
    factory.CreateInstance("CommonTestCases.Factory");
    if (NULL == factory)
        {
        return;
        }

    TestCases::IUrlCanonicalizationTestCasesPtr testCases(
        factory->UrlCanonicalizationTestValues());
    if (NULL == factory)
        {
        return;
        }

    testCases->EncodeUrl(L"http://www.windevnet.com/wdn/current", 3
        , VARIANT_FALSE, 1, 1, TestCases::EncodingUtf8);
    for (unsigned long currentUrl = 1; currentUrl <= testCases->Count;
                    ++currentUrl)
        {
        // do something with testCases->Item[currentUrl].bstrVal;
        }

    return true;
    }
```


Unicode Character Encodings

The Unicode (UCS-2 Unicode, to be specific) encoding is much simpler than the UTF-8 encoding: Simply convert the character's hexadecimal Unicode value to a four character string, then prepend "%u". Thus a period would be "%u002E", and the Lira sign would be "%u20A4".

You're not out of the woods just yet, however. The lower and upper ASCII characters also have full-width variants that live in the 0xFF00 through 0xFFEF range. A full-width character is just another Unicode character, however, so the full-width period would be "%uFF0E". The `CUnicodeMapper` class (download the full code) populates a vector with the standard-to-full-width mapping for each character; its `GetCharacterFromMap` function takes a character (which can be either standard or full width) and returns the matching standard or full-width character. (UTF-8 can also encode full-width Unicode characters.)

Using the Encoders

We've already covered user authentication, path navigation injection, and the specifics of how the UTF-8 and Unicode encoding schemes work, but I've mentioned only obliquely how the encoding schemes are applied to the URL and what the encoding level means in this context. Recall that not only can a plain character be encoded, but that the encoded form can itself be encoded, ad infinitum. The `CCharacterEncoder` class uses a specific encoder (i.e., null—which does nothing, Unicode, or UTF-8) to encode a single character. If given a nonzero encoding level, it also encodes the character multiple times as specified. (No upper limit is imposed, but as just a level of three or four will generate many hundreds of test cases, you generally won't want to go too high.) Two variants are stored at each encoding level: a partial encoding, where only one character in the string is encoded; and a full encoding, where every character in the string is encoded. Thus, encoding a backslash using the UTF-8 encoder might generate the encodings in Table 2.

`CStringEncoder` (Listing 7 is its header; Listing 8 is its body) builds on `CCharacterEncoder` and is typical of how all of the encoders work. Arguments to the constructor include the string to encode, the encoding level, and the encoder to use. `CreateSingleCharacterEncodings` picks a random index into the string to encode, generates encodings for the character at that index, then creates a version of the original string where the source character is replaced with the generated full and partial encodings of the character. `CreateEntireStringFullEncodings`, on the other hand, encodes every character

Handling Illegal Filenames

WHILE NOT really a URL canonicalization issue, another area most applications fail to handle correctly is illegal filenames. Most file-handling APIs allow COM1 and other system-reserved names to be treated identically to a normal file. This is nifty when you need to do so but is a tailor-made security hole when you don't: Since there won't be any data at that port to read, the API you call will patiently wait for some data to appear, which means your app is now effectively hung. Rather than remembering all 23 reserved names, use CommonTestCases.IllegalFilenameTestCases. Not only do you get a list of the reserved names, but you get several of their variants as well: leading and trailing spaces, and upper, lower, and mixed case. However, going through these cases is not enough: You also need to append ".ext" and ".ext.ext" (replacing "ext" with whatever the appropriate extension is for your application), as all these variants are illegal as well.

—M.J.H.

in the string at every level. This will quickly generate extremely long strings, so you can test for buffer overruns at the same time you're testing URL handling. Finally, `CreateEntireStringRandomEncodings` also encodes every character in the string, but rather than always using the encoding for the current level, it randomly selects an encoding level. Thus, the resulting strings will contain unencoded characters, singly encoded characters, doubly encoded characters, and so on.

Generating the Test Cases

Listing 9 shows how to generate and use the test cases. The action starts with `EncodeUrl`. `EncodeUrl` saves off the source URL, the encoding level (more on that in a moment), whether it should generate normal or full-width Unicode, whether it should generate overlong UTF-8 and if so how overlong to go, and the maximum number of characters to prepend to IP addresses. Then it calls `EncodeUrlUsingEncoder` once for each encoder (null, Unicode, and UTF-8). `EncodeUrlUsingEncoder` (Listing 10) wraps all the different encodings and URL manglings we have discussed, mixes in a few more, then spits out anywhere from one to hundreds of test cases.

The first thing `EncodeUrlUsingEncoder` does is add the raw URL to the collection of test cases, passing it through a `CUrl` to ensure it is fully canonicalized. If an encoding level of zero was specified, nothing else happens.

If the requested encoding level was one or higher, the raw URL will be run once through `UrlEscape` to encode any "dangerous" characters. MSDN defines "dangerous" as "those characters that may be altered during transport across the Internet, [which] include the (<, >, ", #, {, }, |, \, ^, ~, [,], and ') characters" (see References). This is the most common form of altered URLs you will encounter.

Requesting an encoding level of two or higher will get you all the encodings we've talked about: User authentication will be inserted, path navigation will be injected, and the IP address will be encoded and collapsed. In addition, various special characters (e.g., dots in the domain name and IP address, dots and slashes in the path), random characters in the domain name and path, the entire domain name, and the entire path will be encoded.

Listing 10 EncodeUrlUsingEncoder

```
void CUrlCanonicalizationTestCases::EncodeUrlUsingEncoder(IEncoder & encoder)
    {
    //-----------------------------------------------
    // All levels get the (fully canonicalized) source URL.
    AddUrl(sourceUrl->Protocol(), sourceUrl->Domain(), sourceUrl->Path());

    //-----------------------------------------------
    // Levels 1 or higher get the URL with "unsafe" (as defined by UrlEscape)
    //    characters encoded.
    if (1 <= encodingLevel)
        {
        AddUrl(sourceUrl->Protocol()
            , EscapeUnsafeCharacters(sourceUrl->Domain())
            , EscapeUnsafeCharacters(sourceUrl->Path()));
        }

    //-----------------------------------------------
    // Levels 2 or higher get the URL with these various additions encoded to
    //    the specified level.  HOWEVER, the encoders don't have the concept of
    //    "escape unsafe characters" (our level 1) -- they just strictly encode
    //    whatever character/string is given them -- so we must reduce the
    //    encoding level we give them by one.
    if (2 <= encodingLevel)
        {
        long encodersEncodingLevel(encodingLevel - 1);

        //-----------------------------------------------
        // Authentication data prepended to the source URL, '@' encodings.
        CCharacterEncoder atEncoder(L'@', encodersEncodingLevel, encoder);
        for (unsigned long level = 0; level <= encodersEncodingLevel; ++level)
            {
            AddUrl(sourceUrl->Protocol()
                , L"username:password" + atEncoder.PartialEncoding(level)
                    + sourceUrl->Domain()
                , sourceUrl->Path());
            AddUrl(sourceUrl->Protocol()
                , L"blahblah" + atEncoder.PartialEncoding(level)
                    + sourceUrl->Domain()
                , sourceUrl->Path());
            AddUrl(sourceUrl->Protocol()
                , L"www.evil.com/You/Are/So/Hacked.htm"
                    + atEncoder.PartialEncoding(level) + sourceUrl->Domain()
                , sourceUrl->Path());
            AddUrl(sourceUrl->Protocol()
                , L"username:password" + atEncoder.FullEncoding(level)
                    + sourceUrl->Domain()
                , sourceUrl->Path());
            AddUrl(sourceUrl->Protocol()
                , L"blahblah" + atEncoder.FullEncoding(level)
                    + sourceUrl->Domain()
                , sourceUrl->Path());
            AddUrl(sourceUrl->Protocol()
                , L"www.evil.com/You/Are/So/Hacked.htm"
                    + atEncoder.FullEncoding(level) + sourceUrl->Domain()
                , sourceUrl->Path());
            }

        //-----------------------------------------------
        // Domain's IP address variants.
        CIPEncoder ipEncoder(sourceUrl->Domain()
            , maxCharactersPrependedToIPAddresses, encodersEncodingLevel
            , encoder);
        for (unsigned long ipEncoding = 1;
            ipEncoding <= ipEncoder.Count(); ++ipEncoding)
            {
            AddUrl(sourceUrl->Protocol(), ipEncoder.Item(ipEncoding)
                , sourceUrl->Path());
            }

        //-----------------------------------------------
        // Domain name encodings.
        CStringEncoder domainEncoder(sourceUrl->Domain()
            , encodersEncodingLevel, encoder);
        for (unsigned long domainEncoding = 1;
            domainEncoding <= domainEncoder.Count(); ++domainEncoding)
            {
            AddUrl(sourceUrl->Protocol(), domainEncoder.Item(domainEncoding)
                , sourceUrl->Path());
            }

        //-----------------------------------------------
        // Dots and slashes in the path encodings.
        CPathEncoder dotEncoder(L'.', sourceUrl->Path(), encodersEncodingLevel
            , encoder);
        for (unsigned long dotEncoding = 1; dotEncoding <= dotEncoder.Count();
            ++dotEncoding)
            {
            AddUrl(sourceUrl->Protocol(), sourceUrl->Domain()
                , dotEncoder.Item(dotEncoding));
            }
        CPathEncoder slashEncoder(L'/', sourceUrl->Path()
            , encodersEncodingLevel, encoder);
        for (unsigned long slashEncoding = 1;
            slashEncoding <= slashEncoder.Count(); ++slashEncoding)
            {
            AddUrl(sourceUrl->Protocol(), sourceUrl->Domain()
                , slashEncoder.Item(slashEncoding));
            }
        CPathEncoder backslashEncoder(L'\\', sourceUrl->Path()
            , encodersEncodingLevel, encoder);
        for (unsigned long backslashEncoding = 1;
            backslashEncoding <= backslashEncoder.Count(); ++backslashEncoding)
            {
            AddUrl(sourceUrl->Protocol(), sourceUrl->Domain()
                , backslashEncoder.Item(backslashEncoding));
            }

        //-----------------------------------------------
        // Partial and full path encodings.
        CStringEncoder pathEncoder(sourceUrl->Path(), encodersEncodingLevel
            , encoder);
        for (unsigned long pathEncoding = 1;
            pathEncoding <= pathEncoder.Count(); ++pathEncoding)
            {
            AddUrl(sourceUrl->Protocol(), sourceUrl->Domain()
                , pathEncoder.Item(pathEncoding));
            }

        //-----------------------------------------------
        // Inject path navigation.
        CNavigationInjector navigationInjector(sourceUrl->Path());
        for (unsigned long injectedPath = 1;
            injectedPath <= navigationInjector.Count(); ++injectedPath)
            {
            AddUrl(sourceUrl->Protocol(), sourceUrl->Domain()
                , navigationInjector.Item(injectedPath));
            }
        }
    }
```

Verifying URL Handling Failures

If you've jumped ahead of the rest of the class and tried these test cases on Internet Explorer (I generally use IE as a litmus test for whether a particular URL variant is valid), you've probably found that some of them don't work. Does this mean the encoding is incorrect? No, it just means IE doesn't handle that particular URL. Some of the time this is intended behavior; IE5 and later don't support dotless IP addresses, for example. Other times it's unclear why it doesn't work (I've found neither Internet Explorer nor any of the Win32 canonicalization APIs like Unicode-encoded URLs, for example). Regardless, the result is you have to pick through a large number of test cases and identify the ones that don't work—and you have to do so on every combination of OS and Internet Explorer version you intend to support. To make that task easier, I've included `UrlVerifier` in the online source code.

`UrlVerifier` does two things: It generates a set of test cases by passing its command-line arguments on to `UrlCanonicalizationTestCases`, then it determines whether each test case is valid (see the sidebar entitled "Test Cases Everywhere" for an explanation of why I wrapped `EncodeUrlUsingEncoder` in `UrlCanonicalizationTestCases` rather than using it directly). To do so, it first repeatedly passes the URL through a canonicalizing function until it stops changing. (The most common URL canonicalization mistake is to only do one pass through the canonicalization function.) Hopefully, I've convinced you that rolling your own such function would be rather complicated; in fact, there's no reason to do so as Windows provides not one, not two, but three separate canonicalizing APIs: `InternetCanonicalizeUrl`, `UrlCanonicalize`, and `UrlUnescape` API functions. Each one takes slightly different options and works in a slightly different manner, but as `UrlVerifier`'s output show, `UrlUnencode` seems to do the best job.

After canonicalizing each URL using each of the canonicalization functions, `UrlVerifier` attempts to download the presumably decoded file using `URLDownloadToCacheFile`. The results of each operation are written to a log file (see Table 3 for a portion of the output from running `UrlVerifier` on http://www.windevnet.com/wdm/default.html) from which you can easily determine which cases Windows thinks are valid and, thus, which cases your application should handle. However, if you use different APIs than `UrlVerifier`, be sure to run your own tests with those APIs rather than taking `UrlVerifier`'s word on which variations should work. And if you're rolling your own URL support, be especially vigilant when deciding which cases to handle.

Summary

There are many different ways to mangle a URL, but the solution to handling all of these is to run every URL through `InternetCanonicalizeUrl` over and over until it stops changing—and to do so the moment you're handed the URL, before you've made any decisions about the URL or taken any action based on it. The Internet can be a wild and wooly place, but with the help of a few simple API calls, you can effectively tame it. The sidebar "Handling Illegal Filenames"

Table 2
Encoding a backslash

Level	Encoding Type	Result
0	Partial	\
	Full	\
1	Partial	%20
	Full	%20
2	Partial	%2520(encoding the % in the level 1 encoding)
	Full	%25%32%30 (encoding the %, 2, and 0)
3	Partial	%252520 (encoding the % in the level 2 partial encoding)
	Full	%25%32%35%25%33%32%25%33%30 (encoding each character in the level 2 full encoding)

Table 3
Sample output from UrlVerifier

Raw URL	UrlUnescape Decoded	UrlUnescape Downloaded
http://www.windevnet.com/wdn/current/	http://www.windevnet.com/wdn/current/	TRUE
http://username:password@www.windevnet.com/wdn/current/	http://username:password@www.windevnet.com/wdn/current/	TRUE
http://blahblah@www.windevnet.com/wdn/current/	http://blahblah@www.windevnet.com/wdn/current/	TRUE
http://www.evil.com/You/Are/So/Hacked.htm@www.windevnet.com/wdn/current/	http://www.evil.com/You/Are/So/Hacked.htm@www.windevnet.com/wdn/current/	FALSE
http://66.35.216.85/wdn/current/	http://66.35.216.85/wdn/current/	TRUE
http://0x42.35.216.85/wdn/current/	http://0x42.35.216.85/wdn/current/	TRUE
http://16931.216.85/wdn/current/	http://16931.216.85/wdn/current/	FALSE
http://www.windevnet.com/wdn/current/./	http://www.windevnet.com/wdn/current/./	TRUE
http://www.windevnet.com/wdn/current/../current/	http://www.windevnet.com/wdn/current/../current/	TRUE
http://username:password%u0040www.windevnet.com/wdn/current/	http://username:password%u0040www.windevnet.com/wdn/current/	FALSE
http://username:password%40www.windevnet.com/wdn/current/	http://username:password@www.windevnet.com/wdn/current/	TRUE
http://www.win%64evnet.com/wdn/current/	http://www.windevnet.com/wdn/current/	TRUE
http://www.win%2564evnet.com/wdn/current/	http://www.windevnet.com/wdn/current/	TRUE
http://www.win%25%36%34evnet.com/wdn/current/	http://www.windevnet.com/wdn/current/	TRUE
http://%77%77%77%2e%77%69%6e%64%65%76%6e%65%74%2e%63%6f%6d/wdn/current/	http://www.windevnet.com/wdn/current/	TRUE
http://www.windevnet.com/wd%%36e/current/	http://www.windevnet.com/wdn/current/	TRUE
http://www.windevnet.com%2f%77%64%6e%2f%63%75%72%72%65%6e%74%2f	http://www.windevnet.com/wdn/current/	TRUE
http://www.windevnet.com%2f%25%37%37%64%6e%%32f%25%36%33%2575%%372%72e%6e%7%34/	http://www.windevnet.com/wdn/current/	TRUE

discusses another test case generator I've provided to help tame another area rife with pitfalls: illegal filenames.

Acknowledgment

Thanks much to Lawrence Landauer, whose patient explanation of how UTF-8 works and assistance with the UTF-8 encoding algorithm was invaluable!

References

Microsoft Security Bulletin MS98-016. http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/ms98-016.asp.

Microsoft Security Bulletin MS01-051. http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/ms01-051.asp.

Boost Random Number Library. http://www.boost.org/libs/random/index.html.

Boost Header lexical_cast. http://www.boost.org/libs/conversion/lexical_cast.htm.

"How to Obscure Any URL." http://www.pc-help.org/obscure.htm.

Writing Secure Code, pp. 323–324, Michael Howard and David LeBlanc. Microsoft Corporation, 2002.

RFC 2279. http://www.ietf.org/rfc/rfc2279.txt.

Microsoft Security Bulletin MS00-057. http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/ms00-057.asp. **w::d**

| Download code › **windevnet.com/wdn/code/** |

Test Cases Everywhere

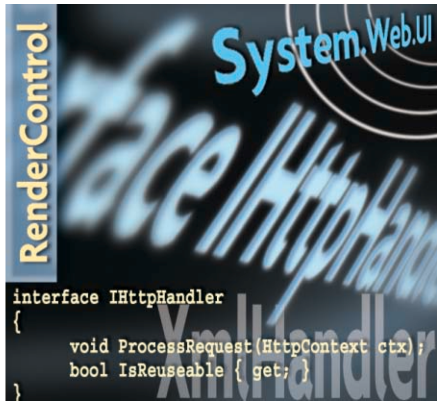
IF YOU'VE done any amount of API testing, you've probably built up a list of standard values you use for various datatypes. I always run through the minimum and maximum value, negative one, zero, and positive one anytime I have a numeric value, for example, and for strings I go through the ASCII characters and a couple really long strings. These values don't change very often, so it may seem that hard-coding them everywhere is not so bad. I've found they change often enough (generally because I'm adding additional values), however, that updating all those hard-coded values is a big time sink.

My solution is to encapsulate these values in a collection I can iterate through in my tests. When I add a new value, I only need to add it to one place and then it's automatically picked up by all my tests. This makes it dead simple for others to use the test cases in their tests as well.

—M.J.H.

Extend ASP.NET Apps Using HttpHandlers

Create custom handlers to perform advanced request processing functions for your web-based applications



THE MAJORITY OF DEVELOPERS using ASP.NET think of the web development framework as a page-oriented model, where an application consists of a series of pages that users interact with to store and retrieve information and to initiate processes on the server. Many developers build these applications with little regard for the inner workings of the ASP.NET processing model and the supporting infrastructure. In fact, there is extensive support in ASP.NET's HTTP pipeline based on a set of classes and interfaces that abstract the HTTP protocol, enabling developers to use the ASP.NET infrastructure to harness the power of the underlying protocols to build powerful web-based applications. This same infrastructure provides the functionality that is used for ASP.NET development and web services in the .NET Framework. HttpHandlers are used by ASP.NET to dispatch HTTP requests to user-defined code, and can be used to develop web applications without the use of .aspx files and the typical code-behind model for ASP.NET development, enabling developers to build and extend web-based applications outside of the page-oriented development model. This article will look at the HTTP pipeline and explore the functionality of HttpHandlers.

The HTTP pipeline support in ASP.NET includes important classes and interfaces such as `HttpContext`, classes that implement `IHttpModule`, and classes that implement `IHttpHandler`. The `HttpContext` class represents the current HTTP request. The classes implementing `IHttpModule` support the pre- and post-processing of HTTP requests, supporting the filtering and modifications of request and response messages in the HTTP pipeline. The `IHttpHandler` interface supports the implementation of classes that can service HTTP requests and provide response output. An example of a class derived from `IHttpHandler` is the `Page` class of the `System.Web.UI` namespace. For a typical ASP.NET application developed using Visual Studio .NET, the classes created in the code-behind model for your ASP.NET application are derived from the `Page` class itself, as shown in the following lines of code:

```
public class WebForm1 : System.Web.UI.Page
{
    private void Page_Load(object sender,
      System.EventArgs e)
    {
// Put user code to initialize the page here
    }
    ...
}
```

This code is automatically generated by Visual Studio .NET for any new ASP.NET project. As you can see, the `WebForm1` class is derived from the `Page` class. `WebForm1` is the code-behind class for the WebForm1.aspx file generated by the IDE. The WebForm1.aspx file includes the following `Page` directive:

```
<%@ Page language="c#"
  Codebehind="WebForm1.aspx.cs"
  AutoEventWireup="false"
  Inherits="WebApplication1.WebForm1" %>
```

This `Page` directive defines a number of attributes for the ASP.NET web form application, including the code-behind class that the application is to inherit from. In this example, the `Inherits` attribute of the directive refers to the `WebForm1` class, which is derived from the `Page` class. When developing a web forms application using ASP .NET, the "under the hood" processing ensures that the forms application corresponds to a class library that is derived from the `Page` class, which implements `IHttpHandler`. In essence, all ASP.NET applications are driven by an underlying implementation of an HttpHandler.

The default configuration for an ASP.NET application is defined by the machine.config file for the .NET Framework installation, and includes the mappings of some standard handlers to certain file extensions. The following entry can be found under the `<httpHandlers>` element of the machine.config file:

```
<add verb="*"path="*.aspx"
 type="System.Web.UI.PageHandlerFactory" />
```

The file entry in machine.config maps the .aspx file extension to the `PageHandlerFactory` class, which is an HTTP handler factory class that can compile the source code referenced in an .aspx file into a class derived from the `Page` class in `System.Web.UI`. Since the `Page` class implements the `IHttpHandler` interface, the instance of the object that is instantiated by the page request can be processed in ASP.NET's HTTP pipeline. This configuration is also used to support the mapping of other extensions, such as the .asmx extension used for ASP.NET web services. Many other file extensions are mapped to various handlers by default at the machine configuration level.

HttpHandler Architecture

Along with the support for standard handlers in ASP.NET, there is functionality to build and extend applications using custom handlers. To build a custom handler class it must be derived from the `IHttpHandler` interface, as shown by the following lines of code:

```
interface IHttpHandler
{
      void ProcessRequest(HttpContext ctx);
      bool IsReuseable { get; }
}
```

RANDY HOLLOWAY *is the founder of Winformation Systems, a technology consulting and training initiative specializing in the development of enterprise systems in the Windows environment and web services technologies. Contact Randy at articles@winformationsystems.com.*

Figure 1 Configure the application's virtual directory to map the extension in IIS to the aspnet_isapi.dll

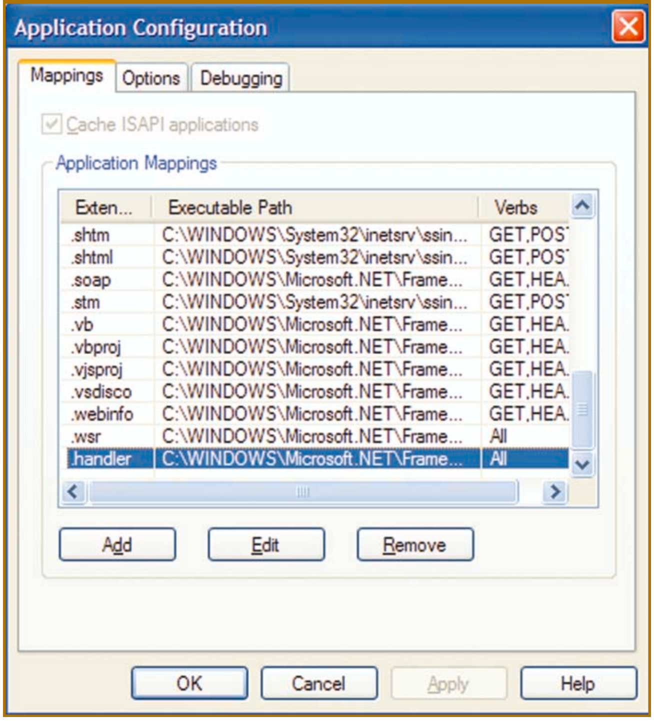


Figure 2 The extension mapping for an IIS application is made to a specific executable and can be limited to specific verbs

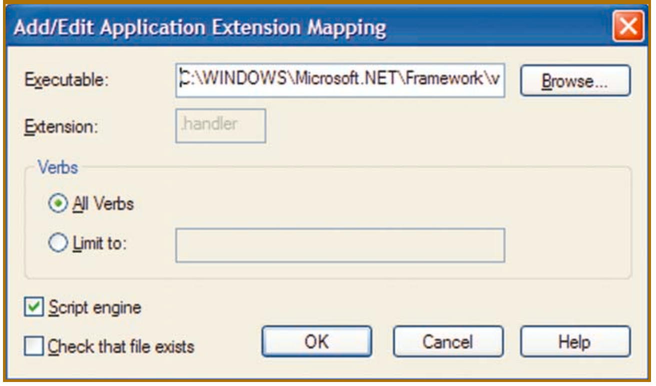
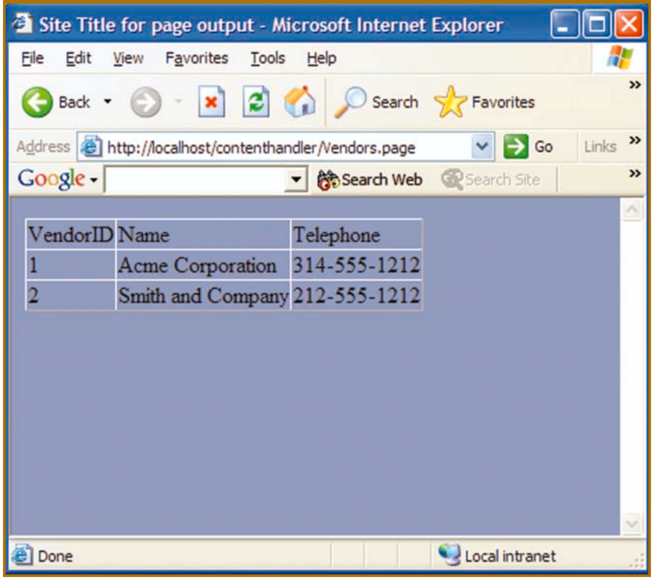


Figure 3 A custom Page class implementation provides all of the functionality of the ASP.NET web forms model without requiring you to use code-behind development



The `ProcessRequest` method is called to process the current request and provide a response. The `IsReuseable` property defines whether or not the handler can be used by more than one request simultaneously. The following code demonstrates the implementation of a sample of a handler class:

```
public class MyHandler : IHttpHandler
{
    public void ProcessRequest(HttpContext ctx)
    {
        ctx.Response.Write("Response from MyHandler.");
    }
    public bool IsReusable
    {
        get { return true; }
    }
}
```

To map an incoming request to a handler, the Web.config file for the application's virtual directory can be used to configure handlers for that application. For the sampler handler class just mentioned, the Web.config file would be modified to include the following section:

```
<configuration>
     <system.web>
      <httpHandlers>
       <add verb="GET" path="*.handler"
     type="Handlers.MyHandler, Handlers"/>
      </httpHandlers>
     </system.web>
</configuration>
```

This configuration file addition adds the support for an HttpHandler supporting a GET request with the extension .handler. This request will then be processed by `Handlers.MyHandler`, located in the `Handlers` assembly. For this custom handler to work, there are a couple of deployment and configuration details to cover. First, the assembly containing the handler class must be located in the bin subdirectory of the virtual directory for the web application, or it must be installed in the GAC for the ASP.NET worker process to be able to locate it. Second, for a handler to be invoked for a particular file extension, that extension must also be mapped to the aspnet_isapi.dll in the IIS application mappings settings, as shown in Figure 1. The extension must be mapped to an executable or library file as shown in Figure 2. The application mapping can also be set up to validate the existence of the file requested for the extension. For the sample URL http://localhost/VDIR/my.handler, the request would be handled by the `MyHandler` class. The path attribute of the add element under `<httpHandlers>` can have a value assigned at the extension level, as shown by the path="*.handler" value, or it can be mapped a specific path such as path="my.handler".

Building Custom HttpHandlers

For many web applications, such as those focused on dynamic delivery of content from disparate data sources like relational databases, XML files, and serialized object stores, the development and maintenance of a traditional page-oriented web site can be tedious and time

Listing 1 XML Handler Class

```
public class XmlHandler : IHttpHandler
{
    public void ProcessRequest(HttpContext ctx)
    {
        string fileRequested =
            ctx.Request.PhysicalPath.ToString();

        //Create instance of the HtmlTextWriter to
        // be populated by DataGrid's HTML output
        StringWriter sw = new StringWriter();
        HtmlTextWriter htmlTW = new
            HtmlTextWriter(sw);

        //Create DataSet and read in XML file
        DataSet ds = new DataSet("XML");
        ds.ReadXml(fileRequested);

        //Create DataGrid and bind to DataSet
        DataGrid outputGrid = new DataGrid();
        outputGrid.DataSource = ds;
        outputGrid.DataBind();

        //Render DataGrid output to HTML
        outputGrid.RenderControl(htmlTW);

        //Provide response output using the
        // StringWriter instance utilized by the
        // HtmlTextWriter constructor
        ctx.Response.Write(sw.ToString());

        ctx.Response.End();
    }

    public bool IsReusable
    {
        get { return true; }
    }
}
```

consuming. With the `HttpHandler` architecture and HTTP pipeline support of ASP.NET, you can build a flexible application while avoiding traditional page-oriented web development.

To demonstrate the power of the `HttpHandler` architecture, I'll walk through the development of a simple handler application that processes requests that are mapped to specific files and are processed by custom handler classes. The first handler will be mapped to the .xml file extension, and will process the file name of the requested URL, load the corresponding XML document into a DataGrid, and render the HTML output of the DataGrid to the `Response` instance. The code for the .xml extension handler is shown in Listing 1. When this handler is executed, the `RenderControl` method of the DataGrid populates the `HtmlTextWriter` object with its HTML output. The `StringWriter` instance passed to the `HtmlTextWriter`'s constructor is then passed to the `Response`'s `Write` method, writing the output back to the browser. To test the handler, deploy it following the process previously outlined and use the following Vendors.xml file to test the process:

```
<Vendors>
    <Vendor>
        <VendorID>1</VendorID>
        <Name>Acme Corporation</Name>
        <Telephone>314-555-1212</Telephone>
    </Vendor>
    <Vendor>
        <VendorID>2</VendorID>
        <Name>Smith and Company</Name>
        <Telephone>212-555-1212</Telephone>
    </Vendor>
</Vendors>
```

Using the `XmlHandler` class from Listing 1, you can write this output in the form of an HTML table back to the browser, displaying the contents of the XML file in a grid form.

With HttpHandlers, there are some other interesting ways that you can handle the request and process the output to a requesting client. By modifying the `XmlHandler` class previously demonstrated, you can process requests for the XML files on your server and provide the output as Excel. In this example, the .xls extension used for Excel files will be mapped to the new handler class. You can then use a URL to request a file that's in XML form on the web server, such as http://localhost/ContentHandler/Vendor.xml, but replace the .xml file extension with .xls. This type of request filtering and processing is ideally performed using an `HttpModule`, but to simplify this example, the logic to modify the file extension will be included in our HttpHandler. The following lines of code show the modified `ProcessRequest` function from the class:

```
public void ProcessRequest(HttpContext ctx)

{

    ...

   //Manipulate Request.PhysicalPath
   //string value to replace file extension
     fileRequested = fileRequested.Replace
                     (".xls", ".xml");

     ...

     //Set ContentType for HttpResponse
     ctx.Response.ContentType =
       "application/vnd.ms-excel";

     //Provide response output using the
     //StringWriter instance utilized by
     //the HtmlTextWriter constructor
     ctx.Response.Write(sw.ToString());

     ctx.Response.End();
}
```

The modified function replaces the file extension in the string that maps the path of the requested file on the server, and then the XML file is processed just as it was in the previous `XmlHandler` example. The file is processed by the handler, loaded into a DataGrid, and rendered to HTML. But prior to writing the response output to the browser, the `ContentType` property of the `Response` instance is changed to Excel. This causes the browser to detect the Excel content type and then prompts the user to save the output to an Excel file or to view the output using Excel within the browser.

Rendering Pages Dynamically

While the two handler classes previously demonstrated support of the processing and output of XML data to the browser without the use of .aspx pages, the output has been different than what you would see on most web sites, since the output hasn't been in a traditional web page form. When a request is processed for the Vendors XML file using the `XmlHandler` from the previous example and the source from your browser is viewed, you see the following output:

```
<table cellspacing="0" rules="all" border="1"
id style="border-
    collapse:collapse;">
    <tr>

<td>VendorID</td><td>Name</td><td>Telephone</td
>
    </tr><tr>
        <td>1</td><td>Acme
Corporation</td><td>314-555-1212</td>
    </tr><tr>
        <td>2</td><td>Smith and
Company</td><td>212-555-1212</td>
    </tr>
</table>
```

Note that the source is raw HTML for the table formatted by the DataGrid and does not conform to HTML standards for a web page. This is not suitable for building a web application that provides the user with a good visual experience or extends functionality beyond the delivery of static content. That's where the `Page` class from the `System.Web.UI` namespace becomes very useful. By creating an instance of the `Page` class, you can build a web page output with controls, page formatting, and other dynamic elements all from within a server-side assembly. This `Page` object can then be rendered to a client browser using HttpHandlers, eliminating the need for .aspx files. The code in Listing 2 demonstrates the use of the `Page` class. The `SimplePage` class overrides the `Render` method and builds on the previous examples by processing an XML file into a DataGrid. The `Page` class can be implemented in an HttpHandler as demonstrated by the following code:

```
public void ProcessRequest(HttpContext ctx)
{
        string fileRequested =
          ctx.Request.PhysicalPath.ToString();

       //Manipulate Request.PhysicalPath
string
       // value to replace file extension
```

```
        fileRequested =
          fileRequested.Replace(".page",
          ".xml");

        //Create instance of the
        // HtmlTextWriter to be populated by
        // DataGrid's HTML output
        StringWriter sw = new StringWriter();
        HtmlTextWriter htmlTW = new
          HtmlTextWriter(sw);

        SimplePage myPage = new
          SimplePage(fileRequested);

        //Render HTML for page object
        myPage.RenderControl(htmlTW);

        //Provide response output using the
        // StringWriter instance utilized by
        // the HtmlTextWriter constructor
        ctx.Response.Write(sw.ToString());
}
```

Using the `RenderControl` method of the custom `Page` class, the output of the page can be written to the `Response` instance. By mapping the .page extension to a new handler, you can extend the previous example to process the XML content and provide the output in a web page format, as shown in Figure 3. Although this example is very simple, by deriving from the `Page` class you can build a `Page` output that takes advantage of the extensive support for controls, page formatting using CSS classes, and other programmatic HTML rendering features.

By using the extensive customization support in the HTTP pipeline for ASP.NET, including the support for HttpHandlers, developers can build web-based applications without using the traditional page-oriented model for web site development. Developers can leverage `System.Web.UI.Page` class implementations and the extensive ASP.NET customization features so that feature-rich applications can be designed and deployed without relying on the ASP.NET code-behind model that's supported by Visual Studio .NET. These features allow developers to better build and deploy web-based applications without the fragility of hand coding and tweaking HTML, while also providing a framework to easily manage the processing of dynamic content from a variety of sources in your web applications. **w::d**

| Download code › **windevnet.com/wdn/code/** |

Listing 2 The SimplePage class

```
public class SimplePage : System.Web.UI.Page
{
    private const string htmlStart =
        "<html><head><title>Site Title for page
        output</title></head><body
        bgColor='#868db9'>";

    private const string htmlEnd =
        "</body></html>";

    private string fileRequested;

    // SimplePage constructor
    public SimplePage(string RequestUrl)
    {
        fileRequested = RequestUrl;
    }

    // Overridden Render method.
    protected override void
        Render(HtmlTextWriter htmlTW)
    {
        //Create DataSet and read in XML file
        DataSet ds = new DataSet("XML");
        ds.ReadXml(fileRequested);

        //Create DataGrid and bind to DataSet
        DataGrid outputGrid = new DataGrid();
        outputGrid.DataSource = ds;
        outputGrid.DataBind();

        // Render the HTML.
        htmlTW.Write(htmlStart);

        //Render DataGrid output to HTML
        outputGrid.RenderControl(htmlTW);

        // Render the HTML.
        htmlTW.Write(htmlEnd);

        // Render the page contents.
        base.Render(htmlTW);
    }
}
```

Licensing in .NET

Building a custom license provider for a simple redistributable component

THE MICROSOFT .NET FRAMEWORK defines a licensing model that allows developers to prevent the use of their classes unless a valid license can be granted. This licensing model is particularly well suited to redistributable components, which are typically sold to developers and subsequently redistributed as a part of the licensed developer's application. Noncomponent classes can easily take advantage of the infrastructure as well.

In this article, we'll enable licensing support for a simple redistributable component. We'll investigate the .NET Framework components involved in licensing, and see how Visual Studio .NET automates much of the licensing process for components. Finally, we'll write code enabling custom license key validation, as well as support for evaluation licenses.

Licensing a Component

Using the .NET licensing model, building a component that requires a valid license key to be present before it can be used is straightforward. You simply add a licensing-specific attribute to your class definition, and add a call to a license validation method from your class constructor.

Listing 1 shows the definition of perhaps the simplest licensed component you can have. The constructor makes a call to `LicenseManager.Validate`, which asks the licensing framework to request a valid license for the class. The down-and-dirty work of finding and validating a license key, and subsequently granting a license, is done by a component called a "license provider." Microsoft supplies a simple license provider, `LicFileLicenseProvider`, which we associate with the component in this example using the `[LicenseProvider]` attribute on the class. We'll look at what's involved with creating custom license providers shortly.

If the license validation is successful, `Validate` simply returns and allows the component to be created and used normally. If, however, a license cannot be granted, `Validate` raises a `LicenseException` exception, preventing the component from being used.

If you build the component given in Listing 1 and then attempt to add it to a host application (by referencing its assembly, adding it to the Visual Studio Toolbox, and dropping it onto the design surface), you'll get the message box shown in Figure 1. As the authors of the licensed component, this is the behavior that we want: Our class requires that a license be granted before it is used, but we haven't yet provided any evidence that one should be granted. Technically, what has happened is that `LicFileLicenseProvider` could not find a valid license key for our component, so a `LicenseException` was raised inside the `LicenseManager.Validate` method.

License providers such as `LicFileLicenseProvider` grant licenses only when a valid license key can be located. A license key is a string, and may be a serial number, an installation code, or any other string recognized by the license provider. Locating a license key is the responsibility of the license provider, and it may do so in whatever manner it chooses. In the case of `LicFileLicenseProvider`, a simple text file is expected to exist for each licensed class, containing nothing but the license key string. The text file needs to have the name <classname>.lic and be located in the same directory as the licensed class's assembly (`<classname>` is the namespace-qualified name of the licensed class). Other license providers could obtain license keys from the registry, a web service method, or however else they might choose.

Validating a license key is the license provider's primary responsibility. `LicFileLicenseProvider` isn't very sophisticated when it comes to this: A valid license key is simply a string of the form "<classname> is a licensed component." We'll see how to override this validation logic soon.

Listing 2 shows the license key (.lic) file for `SimpleLicensedComponent`, which contains just one line:

```
SimpleLicensedComponent is a licensed
  component.
```

Placing this file in the directory containing our simple component's assembly allows it to be used normally, just as if it weren't licensed. The host application does not need to do anything special to create the licensed class—the licensing framework handles it all automatically. (Note that during development of a licensed component, you'll want to have the .lic file for your component in your project's obj\Debug directory, since that is where the component's assembly lives. Place another copy in the obj\Release directory if you're testing release builds of the component as well.)

When the host application is compiled and distributed to another computer, the license key file is not copied along with the component's assembly. You might expect that this would result in a licensing error when the application attempts to create the component at run time. Instead, the component is created

MYK WILLIS *is President of Wanderlust Software LLC, publisher of the licX Licensing Component for .NET. He can be reached through the company web site at http://www.wanderlust-software.com/.*

successfully. The .NET licensing framework, the license provider, Visual Studio .NET, and a utility called the "license compiler" all work together during the build process to "remember" that a license had been granted for the component at design time.

The License Compiler

The .NET Framework SDK ships with a utility called the "license compiler" (lc.exe), which simplifies the distribution of applications that use licensed classes. The license compiler embeds the required license keys into a host application's assembly so that they can be found on whatever machine the host application is run. This obviates the need to distribute license key files with the application.

The license compiler is invoked after a host application has been built. It is provided with the name of the host application's assembly, and a list of the licensed classes that the application uses. The license compiler creates an instance of each of these licensed classes and, in cooperation with the classes' license providers, extracts a run-time license key from each.

The license keys obtained by the license compiler are written to a binary file named <assembly>.licenses, where `<assembly>` is the name of the host application assembly. This .licenses file is then embedded as a resource in the host application's assembly using the services of the .NET assembly linker (al.exe).

It turns out that the plumbing necessary to allow the license compiler access to the license keys is quite involved. Suffice it to say that the license provider is required to call a special method exposed by the license compiler when invoked at design time. `LicFileLicenseProvider` handles this automatically when it (or a class derived from it) is used. Later we'll see an example that needs to deal with this directly.

If you use Visual Studio .NET, you typically don't need to interact directly with the license compiler. That's because Visual Studio automatically tracks the use of licensed components and invokes the license compiler seamlessly during the build process. You can see this by adding a licensed component to your project by dragging it to the designer surface. Visual Studio adds the name of the component class to a file named "licenses.licx" in the host application's project directory (enable the "Show All Files" option in the Solution Explorer to see licenses.licx listed). The classes listed in licenses.licx are passed to the license compiler during the build.

Note that if you create an instance of a licensed class without using the Visual Studio designer (i.e., if the licensed class is not a component, or it is otherwise created nonvisually), you'll need to manually add that class name to the licenses.licx file. If you're not using Visual Studio .NET for development, you'll need to maintain your own licenses.licx file and invoke lc.exe manually as part of your build process. See the .NET Framework SDK documentation in this case for more detailed information on using the license compiler.

Figure 1
Error when trying to use a licensed class without a valid license key

Deriving New License Providers

`LicFileLicenseProvider` is not a very sophisticated license provider, but it is the only one supplied by the .NET Framework. Fortunately, it's easy to derive a new class from `LicFileLicenseProvider` that overrides the default license key validation logic. It's also possible to write a license provider from scratch, which allows for considerable flexibility in implementing sophisticated licensing schemes.

When deriving a new license provider from `LicFileLicenseProvider`, you must override both the `IsKeyValid` and `GetKey` methods. `IsKeyValid` is called to validate a license key string that the base class has located. `GetKey` is called at design time to get the license key string that will be remembered for validation at run time. Listing 3 shows the definition of `DerivedLicenseProvider`, which demonstrates overriding these methods.

In `DerivedLicenseProvider`, we've replaced the base class's implementation of `IsKeyValid` with logic that expects a license string to be composed of two parts: the licensed class's full name and the hash code of a secret string appended to the full name. While far from secure, this scheme makes valid license keys a little less obvious

Listing 1
A simple licensed component

```
using System;
using System.ComponentModel;
using System.Windows.Forms;

[LicenseProvider(typeof(LicFileLicenseProvider))]
public class SimpleLicensedComponent : Component
{
    public SimpleLicensedComponent()
    {
        LicenseManager.Validate( typeof(SimpleLicensedComponent) );
    }
}
```

Listing 2
License key file for use with LicFileLicenseProvider

```
SimpleLicensedComponent is a licensed component.
```

Listing 3
A simple license provider derived from LicFileLicenseProvider

```
using System;
using System.ComponentModel;

public class DerivedLicenseProvider : LicFileLicenseProvider
{
    protected override bool IsKeyValid(string key, System.Type type)
    {
        return key.Equals( GetValidLicenseKey(type) );
    }

    protected override string GetKey(System.Type type)
    {
        return GetValidLicenseKey(type);
    }

    private string GetValidLicenseKey(Type type)
    {
        int hashCode = (type.FullName + "Secret").GetHashCode();
        return String.Format( "{0}:{1,8:X}", type.FullName, hashCode );
    }
}
```

from the outside. GetValidLicenseKey contains the code to generate a valid license key for a class of a given type.

When deriving from LicFileLicenseProvider, overriding GetKey is the method by which you provide (at design time) the license key that will be remembered for use at run time.

You might wonder why it's necessary to implement GetKey. After all, the base class is the one that found the license key in the first place, so can't it just remember it itself? The reason seems to be that while in most cases design-time and run-time license keys will be the same, it's not a strict requirement. You might want to implement GetKey such that the remembered license key contains not just a serial number for your component, but also information about the licensed user, version information from the development machine, or something else. Because the license key string is remembered as a resource embedded in the host application's assembly, inspecting an application that uses your component could reveal the identity of the licensee.

In any case, you must override GetKey when deriving from LicFileLicenseProvider. Otherwise, the default base class implementation will return "xyz is a licensed component," which will probably not be recognized as valid by your custom validation logic.

After building our derived provider, we can test it by changing the [LicenseProvider] attribute on our SimpleLicensedComponent:

```
 [LicenseProvider(
  typeof(DerivedLicenseProvider))]
public class SimpleLicensedComponent ...
```

Because license providers run at design time in the context of Visual Studio, debugging is a little awkward. To do it, load a solution containing your license provider, a component licensed with it, and a host application (i.e., a Windows Forms application) for your component in Visual Studio. Next, open a second instance of Visual Studio and debug the first by using Tools | Debug Processes… from the main menu. You should now be able to set breakpoints at convenient places in your license provider (i.e., at IsKeyValid), which should be triggered when you drag the licensed component onto the design surface of the test host application.

Note that Visual Studio will need to be able to find the license provider's assembly at design time. You'll save yourself a lot of hassles if you include a custom license provider in the same assembly as the class(es) it licenses.

Creating a License Provider from Scratch

Sometimes deriving from LicFileLicenseProvider isn't adequate for implementing the licensing scheme you want to use. For example, if you want to use a registry value to hold your component's license key instead of a license file, you're out of luck with LicFileLicenseProvider. Implementing your own license provider from scratch isn't terribly difficult, but you do have to deal with several issues that we've so far avoided by using LicFileLicenseProvider.

> Locating a license key is the responsibility of the license provider, and it may do so in whatever manner it chooses

A custom license provider must derive from the abstract System.ComponentModel.LicenseProvider class. This class defines just a single method, GetLicense, which needs to be implemented by the derived class. GetLicense is given information about the class for which a license is being requested and the context of the call, and responds by granting a license to the caller.

GetLicense has the following signature:

```
License GetLicense(
 LicenseContext context, Type type,
 object instance, bool allowExceptions
 )
```

The type and instance parameters refer to the object for which a license is being requested. They are the parameters passed to LicenseManager.Validate by the licensed object itself. allowExceptions specifies whether the routine should raise an exception when a failure to grant a license occurs (as opposed to just returning NULL).

Listing 4 A better base class for implementing a custom license provider

```
using System;
using System.ComponentModel;

public abstract class CustomLicenseProvider : LicenseProvider
{
    public override License GetLicense(LicenseContext context, Type type,
        object instance, bool allowExceptions)
    {
        // Step 1: locate the license key
        string licenseKey = null;
        switch ( context.UsageMode )
        {
            case LicenseUsageMode.Designtime:
                licenseKey = GetDesignTimeLicenseKey(type);
                context.SetSavedLicenseKey( type, licenseKey );
                break;
            case LicenseUsageMode.Runtime:
                licenseKey = context.GetSavedLicenseKey( type, null );
                break;
        }
        if ( licenseKey == null )
        {
            if ( allowExceptions ) throw new LicenseException(type, instance,
                                    "No appropriate license key was locat-
ed.");
            else return null;
        }

        // Step 2: validate the license key
        bool isValid = IsLicenseKeyValid( type, licenseKey );
        if ( !isValid )
        {
            if ( allowExceptions ) throw new LicenseException(type, instance,
                                    "License key is not valid.");
            else return null;
        }


        // Step 3: grant a license
        License license = CreateLicense(type, licenseKey);
        if ( license == null )
        {
            if ( allowExceptions ) throw new LicenseException(type, instance,
                                   "license could not be created.");
            else return null;
        }

        return license;
    }

    protected abstract string GetDesignTimeLicenseKey(Type type);
    protected abstract bool IsLicenseKeyValid(Type type, string licenseKey);
    protected abstract License CreateLicense(Type type, string licenseKey);
}
```

The context parameter is more complicated. It refers to a `LicenseContext` object that provides information about whether the class is being created at design time or at run time, and provides utility methods that can be used by the provider.

The return value is of type `System.ComponentModel.License`, which is the abstract class from which all licenses must be derived. A minimal implementation of a `License` class must provide a `Dispose` method and a read-only `LicenseKey` property, but other properties and methods can often be useful. We'll see this in a moment.

You'll save yourself a lot of hassles if you include a custom license provider in the same assembly as the class(es) it licenses

Listing 4 shows a base class for a custom license provider that breaks down the implementation of `GetLicense` into three steps. First, a license key is located. Second, the found license key is validated. Finally, a `License` object is created and returned to the caller. `GetLicense` in `CustomLicenseProvider` coordinates these three steps, each of which is implemented by one of the virtual methods `GetDesignTimeLicenseKey`, `IsLicenseKeyValid`, and `CreateLicense`.

`CustomLicenseProvider` is implemented as an abstract base class in order to deal with the intricacies of design time versus run time, raising exceptions when appropriate and so forth, without dictating the policy for locating, validating, and granting licenses. It thus serves as a more convenient base class than `LicenseProvider`, and a more flexible one than `LicFileLicenseProvider`.

The first part of `CustomLicenseProvider.GetLicense` deals with locating a license key. At design time, a license key should be retrieved from a file, registry key, or other source on the development machine as implemented by `GetDesignTimeLicenseKey`. The license key obtained by this method at design time is saved away for run-time use by the `SetSavedLicenseKey` method of the context parameter.

At design time, the context parameter refers to an object supplied by Visual Studio .NET. This object implements `SetSavedLicenseKey` in such a way as to facilitate the creation of the licenses.licx file and the <assembly>.licenses file that is eventually added as an assembly resource. The context object has a private `System.Collections.Hashtable` containing all of the licenses that have been saved with `SetSavedLicenseKey`, and it is simply by serializing this hash table to disk that the <assembly>.licenses file is created.

When `GetLicense` is called at run time, the context parameter is supplied by the .NET run time. Instead of using `GetDesignTimeLicenseKey` to find the license key on the development machine, our custom license provider uses `context.GetSavedLicenseKey`. `GetSavedLicenseKey` retrieves the license key from the hash table that the run-time context had previously deserialized from the assembly resource. Note that the second parameter to `GetSavedLicenseKey` is NULL to specify that the entry assembly (that is, the assembly used to start the application) should be used; it appears that specifying another assembly for this parameter is not supported.

Despite the underlying complexity of license key serialization, `CustomLicenseProvider` implements its share of the work with just a few lines of code. Using the `UsageMode` property of the `LicenseContext` to indicate whether we are being invoked at design time or run time, the license key is retrieved from the appropriate location:

```
switch ( context.UsageMode )
{
case LicenseUsageMode.Designtime:
  licenseKey =
    GetDesignTimeLicenseKey(type);
  context.SetSavedLicenseKey(
    type, licenseKey );
  break;
case LicenseUsageMode.Runtime:
  licenseKey =
    context.GetSavedLicenseKey(
    type, null );
  break;
}
```

Listing 5 A license object derived from System.ComponentModel.License

```
using System;
using System.ComponentModel;

public class CustomLicense : License
{
    string licenseKey;
    bool isEvaluation;

    public override void Dispose() {}

    public override string LicenseKey
    { get { return licenseKey; } }

    public bool IsEvaluation
    { get { return isEvaluation; } }

    public CustomLicense(string licenseKey, bool isEvaluation)
    {
        this.licenseKey = licenseKey;
        this.isEvaluation = isEvaluation;
    }
}
```

Listing 6 A license provider with support for evaluation license keys

```
using System;
using System.ComponentModel;
using Microsoft.Win32;

public class EvaluationLicenseProvider : CustomLicenseProvider
{
    protected override string GetDesignTimeLicenseKey(Type type)
    {
        RegistryKey regKey = Registry.LocalMachine.OpenSubKey(
                            "Software\\WdmLicensing\\" + type.FullName );
        if ( regKey == null )
        {
            return null;
        }
        return regKey.GetValue( "LicenseKey" ).ToString();
    }

    protected override bool IsLicenseKeyValid(Type type, string licenseKey)
    {
        return ( licenseKey.Equals( GetValidLicenseKey(type) ) ||
                 licenseKey.Equals("Evaluation") );
    }

    private string GetValidLicenseKey(Type type)
    {
        int hashCode = (type.FullName + "Secret").GetHashCode();
        return String.Format( "{0}:{1,8:X}", type.FullName, hashCode );
    }

    protected override License CreateLicense(Type type, string licenseKey)
    {
        return new CustomLicense(licenseKey, licenseKey.Equals("Evaluation"));
    }
}
```

Listing 7 A component that displays a visible message when used in evaluation mode

```
using System;
using System.ComponentModel;
using System.Windows.Forms;

[LicenseProvider(typeof(EvaluationLicenseProvider))]
public class TryAndBuyComponent : UserControl
{
    private Label evaluationLabel;

    private void InitializeComponent()
    {
        this.evaluationLabel = new System.Windows.Forms.Label();
        this.evaluationLabel.Text = "Evaluation Mode";
        this.Controls.Add( this.evaluationLabel );
    }

    public TryAndBuyComponent()
    {
        CustomLicense license;

        InitializeComponent();

        license = (CustomLicense) LicenseManager.Validate(
            typeof(TryAndBuyComponent), this );

        if ( license.IsEvaluation ) evaluationLabel.Visible = true;
        else                        evaluationLabel.Visible = false;

        license.Dispose();
    }
}
```

The second step of `GetLicense` is validation of the license key. This is done using `IsLicenseKeyValid`, which serves essentially the same purpose as the `IsKeyValid` method of `LicFileLicenseProvider`.

The third and final step of `GetLicense` is the actual granting of a license. When we derived from `LicFileLicenseProvider`, the `License` object representing the granted license was created by the base class, and we didn't have to deal with it. Here, we must rely on the `CreateLicense` method to create an appropriate license object given a class type and license key.

Note that at each error path in `GetLicense`, the `allowExceptions` parameter is consulted to determine whether the routine should return NULL or raise a `LicenseException`. While I've never seen a situation where `allowExceptions` is False, the documentation indicates that exceptions should not be raised if it is. Regardless, returning NULL results in the license manager raising a `LicenseException`.

Enabling Evaluation Support

Let's take the base class for the custom license provider shown in Listing 4 and derive a new provider that has support for evaluation license keys. Evaluation license keys allow a class to be used, but may degrade its functionality in some way to encourage the purchasing of a normal license for the class.

There are three methods we need to override when we derive from `CustomLicenseProvider`. For this new provider, I've implemented `GetDesignTimeLicenseKey` to consult a registry value based on the name of the type being licensed. `IsLicenseKeyValid` uses the same simple hash function as our implementation of `IsKeyValid` in the provider we derived from `LicFileLicenseProvider`, except that a string `Evaluation` will be considered valid as well.

To implement `CreateLicense`, we need to create an appropriate license class to represent a granted license. Listing 5 shows the class that we use with our evaluation license provider. The base class requires that we implement the read-only `LicenseKey` property and the `Dispose` method, while the `IsEvaluation` property is new. `IsEvaluation` is set to True when the located license key string is `Evaluation`. Listing 6 shows the complete evaluation license provider.

In order for a licensed class to behave differently based on whether a normal or evaluation license has been granted, it needs some way to get at the `License` object returned from `GetLicense`. This is done using an overload of `LicenseManager.Validate`. If you pass both the class type and an object instance (`this`) to `Validate`, you'll get back the same `License` object that was returned from `GetLicense`.

Listing 7 shows a simple component licensed with our evaluation license provider. `TryAndBuyComponent` uses the `IsEvaluation` property of the `CustomLicense` returned from `LicenseManager.Validate` to show an "evaluation mode" label when an evaluation license key has been used. Another component might disable some of its functionality or otherwise change its behavior in this case. Because `License` derives from `IDisposable`, `TryAndBuyComponent` calls `Dispose` when it is done using it.

Using this same technique, you could allow the full version of your commercial component to be widely distributed, packaged with an evaluation license to degrade its functionality in some way. When a developer tries the evaluation version of your component and decides to purchase it, you simply give them a nonevaluation license key that eliminates the evaluation restrictions. In this way, you don't need to maintain multiple distribution packages for your product.

A more sophisticated licensing scheme that allows for individual features to be enabled or disabled based on the license key used could be built. By adding properties to the `License` object used, the license provider can effectively tell the licensed component which features should be allowed.

Conclusion

Licensing classes in .NET has significant support from the Framework and Visual Studio. While the out-of-the-box licensing support is rather restrictive, it is extensible enough to allow for considerable flexibility in implementing sophisticated licensing schemes. **w::d**

| Download code › **windevnet.com/wdn/code/** |

Please send us your best tricks and hacks—those clever pieces of code to make things work the way they should! You'll receive at least $50 for each tip that we print. Send your submissions to wdletter@cmp.com with the header "Tech Tip submission."

When Shared Data Segments Aren't Shared

BY LARRY HAMILTON
l.hamilton@the-lhsoftware.com

THERE ARE SEVERAL WAYS to share data between instances of DLLs. One of the methods is to use Shared Data segments that are set up in the .def file for the DLL. This method is much easier than going through all the work that memory-mapped files require. There is, however, a potentially time-consuming surprise in store for you if you do not understand one caveat about the shared memory segments.

Shared data segments are only shared between copies of a DLL loaded from the same location in the filesystem. This can be quite surprising when trying to debug a DLL with shared data segments in a system with many programs using the DLL. It is not uncommon to have the DLL loaded by parts of the system from its normal location and to also have the DLL loaded from the Debug directory in the project location. This results in two copies of the shared data segments.

To demonstrate this problem, I have written a very simple DLL that supports reading and writing a simple integer number between multiple applications. The DLL simply defines three access functions and a shared data segment with an integer. The application (included in this month's code archive) shows where the DLL is loaded from and allows the user to read and write integer values. The projects to build the DLL and application have a "Custom Build Step" that copies the target to both the bin and bin2 directories.

By running two copies of TheApp.exe from the bin directory, you can see that writing a value in one instance of the application will cause that value to be read in the other instance, just as would be expected.

Now without closing either of these instances, go to the bin2 directory and start an instance of TheApp.exe from there. Notice that when doing a read from this instance, you do not get the value written by the instances run from bin. As an additional step, you could run another instance of TheApp.exe from bin2 and see that the instances run from the same directory use the same shared memory.

Preventing Class Derivation in Visual C++ .NET

BY EHSAN AKHGARI
Ehsan@BeginThread.com

HERE IS A TIP for preventing derivation from a class in C++. Suppose you have a class `B`, and you don't want any class to derive from this class. C++, unlike other languages such as Java, doesn't have any built-in support for this task, but this can be done using virtual inheritance. You should create and use a `Lock` class as this example shows:

```
class Lock {
        Lock() {}
        friend class B;
};

class B : private virtual Lock
{
public:
        B() {}
};

class D : public B
{
public:
        D() {}
};

int main()
{
        B b;
        D d;

        return 0;
}
```

`B` is a friend of `Lock`, so `B::B()` has access to `Lock::Lock()` and can initialize the virtual base class. On the other hand, `D` is not a friend of `B`, so `D::D()` can't initialize the virtual base class due to its inability to call `Lock::Lock()`. Thus, attempting to compile the aforementioned program will lead to an error. Microsoft Visual C++ .NET 2003 produces the following error messages when compiling this application:

```
h:\Projects\VC++.NET\test\test\test\test.cpp(17)
  : error C2248: 'Lock::Lock' : cannot access
  private member declared in class 'Lock'
h:\Projects\VC++.NET\test\test\test\test.cpp(3)
  : see
declaration of 'Lock::Lock'
h:\Projects\VC++.NET\test\test\test\test.cpp(2)
  : see
declaration of 'Lock'
h:\Projects\VC++.NET\test\test\test\test.cpp(22)
  : error C2248:
'Lock::Lock' : cannot access private member
  declared in class 'Lock'
h:\Projects\VC++.NET\test\test\test\test.cpp(3)
```

GEORGE FRAZIER *is a software engineer in the System Design and Verification group at Cadence Design Systems Inc. and has been programming for Windows since 1991. He can be reached at georgefrazier@yahoo.com.*

```
  : see
declaration of 'Lock::Lock'
h:\Projects\VC++.NET\test\test\test\test.cpp(2)
  : see
declaration of 'Lock'
```

Do/While Macros Souped Up and Revisited (First Movement)

BY CINDY ROSS
rossc@us.ibm.com

IN THE JUNE 2003 issue, John Szakmeister suggested that the macro wrapper (previously presented by Raja Venkataraman) could be improved by removing the `do/while` construct. But the original version with the `do/while` construct is better, as it can be used in any context where a C statement can be used.

For example, using the macro from the previous article, this code compiles fine:

```
#define MY_ASSERT_ONE(x) do { \        if (!(x)) { \
      _asm int 3 \
   } \
   } while (0)

main()
{
   if ( 1 )
      MY_ASSERT_ONE(1);
   else
      return 1;
}
```

but we get a syntax error if we omit the do/while construct:

```
#define MY_ASSERT_ONE(x) { \        if (!(x)) { \
      _asm int 3 \
   } \
   }

main()
{
   if ( 1 )
      MY_ASSERT_ONE(1);
   else
      return 1;
}
```

Do/While Macros Souped Up and Revisited (Second Movement)

BY PETTER HESSELBERG
petter.hesselberg@accenture.com

REGARDING JOHN SZAKMEISTER'S JUNE 2003 Tech Tip ("A Better Macro Wrapper for Visual C++"), which was a response to Raja Venkataraman's February 2003 Tech Tip ("do/while Macros for C++").

Venkataraman's original Tech Tip starts by showing the problems associated with a simplistic ASSERT implementation such as this:

```
#define MY_ASSERT_ONE( x )  if ( !( x ) ) { \
  _asm int 3 \
}
```

Using the aforementioned definition, the following code generates an "illegal `else` without matching `if`" compiler error:

```
if ( x )
    MY_ASSERT_ONE( y );
else
    MY_ASSERT_ONE( z );
```

This is fixable in two ways: Either remove the semicolons (a strange-looking and error-prone solution) or insert braces (which is a good practice in any case):

```
if ( x ) {      MY_ASSERT_ONE( y );
} else {
    MY_ASSERT_ONE( z );
}
```

Venkataraman then goes on to show the MFC/ATL solution, in which a semantically NULL `do/while` loop makes the semicolon required rather than forbidden:

```
#define MY_ASSERT_TWO( x )  do { \
  if ( !( x ) ) { \
        _asm int 3 \
    } while ( 0 )
```

This code is, in any case, wrong because it has two opening braces but only one closing brace. The correct version looks like this:

```
#define MY_ASSERT_TWO( x )  do { \
  if ( !( x ) ) { \
        _asm int 3 \
    } } while ( 0 )
```

Szakmeister's Tech Tip update then "improves" on Venkataraman's by going back to the original version. He does not discuss the semicolon problem, except to note that his version is "harmful if not used correctly"—this makes you wonder if he actually read all of the original Tech Tip.

The standard header file, assert.h, has a different solution:

```
#define assert(exp) \      (void)( (exp) || \
    (_assert(#exp, __FILE__, __LINE__), 0) )
```

This approach doesn't help if you insist on using an inline assembler, though; there's no way of making an assembly statement part of an expression. The closest you get is something like this:

```
int failedAssert( void ) {      __asm int 3
    return 0;
}

#define MY_ASSERT_THREE( x ) \
    (void)( (x) || failedAssert())
```

This works well enough—except that the interrupt occurs in `failedAssert()` rather than at its point of call, and you have to go manually one step up the call stack in the debugger. But inline assembly is not a necessity in Windows programming; there's `DebugBreak()` to the rescue:

```
#define MY_ASSERT_FOUR( x ) \
    (void)( (x) || DebugBreak())
```

But oops—`DebugBreak()` is a void function, and you can't have void as an operand to the `||` operator. Let's try the ternary conditional operator instead:

```
#define MY_ASSERT_FIVE( x ) \
    ((x) ? (void) 0 : DebugBreak())
```

If you absolutely must have inline assembly, I suspect that the do/while technique really is the safest approach—in spite of its clunkiness

The trick to safe macro programming in C/C++ is to avoid the outer braces—you need an expression rather than a statement. Consider the comma operator, also called the "sequential-evaluation operator," which lets you evaluate two or more expressions in a context where only one expression is allowed. This allows cool macros like the following, good for trashing a pointer you've just released (the purpose being to root out dangling pointer errors):

```
#define reset_pointer( p )  \
    ( memset( &( p ), 0xacACacAC, sizeof( p )
  ), \
        assert( 4 == sizeof( p ) ) )
```

This is pretty cool, but can be confusing—the comma operator is easily confused with the comma that separates function or macro arguments; it is emphatically not the same thing. The following example shows how you might call a function with three parameters; the value of the second parameter passed is 3; the value of `b` after the function call is 1:

```
myFunction( a, (b = 1, b + 2), c );
```

If you absolutely must have inline assembly, I suspect that the `do/while` technique really is the safest approach—in spite of its clunkiness.

Editor's note: I hope our reader's have enjoyed this ongoing debate about do/while macros (and I'm sure we'll get the big e-mail gong if not). Kudos to Raja Venkataraman, who started the thread, and to everyone who vigorously threw in their opinion. Have an opinion of your own about any of our tips? Please send them to me at georgefrazier@yahoo.com. **w::d**

| Download code › **windevnet.com/wdn/code/** |

DataGrid Control is a powerful component, but you need to customize to get the most out of it

Create Custom Columns for the WinForms DataGrid Control

THE WINDOWS FORMS DATAGRID control is a rich user-interface component that can be effectively used to enhance the quality of data-bound forms in .NET desktop applications. The control features many capabilities including .NET-style data binding, column binding, plenty of visual settings, automatic navigation, in-place editing, and hierarchical display. It is not perfect, though. In particular, it doesn't provide great support for data types more complex than numbers and strings. For example, dates can only be rendered as strings and a URL stored in a data source is nothing more than a string. Is there a way to improve the rendering for certain types of data? A helpful suggestion on how to proceed comes from the Web counterpart of the control—the ASP.NET `DataGrid` control.

The ASP.NET `DataGrid` supports several types of columns including data-bound, button hyperlink, and templated columns. The Windows Forms `grid` control supplies only two types of columns—textbox and boolean columns. The default type of column is the textbox column. The text associated with the cell is rendered as the text of an optionally readonly textbox control. If the cell contains a boolean value, it is rendered through a checkbox. This is what the control can do for you. If it is not enough, you can only extend it with custom columns.

Why You Need Custom Columns

Suppose you have a database of articles with columns like title, author ID, and the URL to the online article. Unless you take steps to alter it, the `DataGrid` would show its contents as in Figure 1. There are a few problems with that grid. First off, the author ID is displayed as a number whereas the name would have been more appropriate and clearer for the end user. Second, the URL is static text and is not clickable. Third, if the user wished to modify the name of the article's author, wouldn't it be nice if you could serve a list of the available options? In short, all these options are not applicable with built-in columns; a couple of ad hoc columns will do the job.

A custom column is a class that derives from `DataGridColumnStyle`. The base class doesn't implement all the functions required by the grid, so when you start creating a new column class, be prepared to implement a few abstract methods. In addition, a few methods should be conveniently overridden for a better result. Let's review the overall pattern.

Logically speaking, a `DataGrid` column is a collection of cells—that is, independent objects representing a block of the column data. Such a high-level description clashes with the actual implementation of the grid in which the whole content is simply drawn to the underlying window canvas using GDI+ calls. All in all, this is not much different from what happens in Win32 programming in which you need to manually paint the window's background. The only difference is that the tools available for the job are of a slightly higher level. For performance reasons, the `DataGrid` maintains, at most, one instance of a control per column. The lifetime of this control and, more importantly, its type depend on the characteristics and the implementation of the column class.

When the user clicks on a column cell, the class pops up the hidden control and sizes it to fit into the cell bounds. You can choose to create a brand new instance of the control whenever the user clicks (single-call pattern), or can you go for a singleton approach: create the control once and reuse for the duration of the application. Depending on your choice, the number of active controls at a time ranges from 0 (single-call approach, no cell selected) to the number of the displayed columns.

There are particular types of control behind each type of column. The `DataGridTextBoxColumn` class is built around an instance of a textbox control. Instead, a checkbox control is the bread and butter of the `DataGridBoolColumn` class. When you build your own column

DINO ESPOSITO *is Wintellect's ADO.NET and XML expert and is a trainer and consultant based in Rome, Italy. He is a contributing editor to* MSDN Magazine, *writing the "Cutting Edge" column, and is the author of several books for Microsoft Press, including* Building Web Solutions with ASP.NET *and* Applied XML Programming for .NET. *Contact him at dinoe@wintellect.com.*

classes, you can choose the control that best suits your expectations. For example, if you want to architect a column optimized for date values, the `DateTimePicker` control sounds like the perfect choice. Incidentally, the MSDN documentation just contains a sample grid column for dates based on the date-picker control.

Figure 1 Without custom columns, the DataGrid control is not very useful

Table 1 DataGridColumnStyle overridable methods

Method	Abstract	Description
Abort	Yes	Initiates a request to interrupt an edit procedure.
Commit	Yes	Initiates a request to complete an editing procedure.
ConcedeFocus	No	Notifies a column that it must release the focus to the control it is hosting.
Edit	Yes	Prepares the cell for editing a value.
GetMinimumHeight	Yes	Returns the minimum height for a cell.
GetPreferredHeight	Yes	Returns the height used for automatically resizing columns.
GetPreferredSize	Yes	Returns the preferred size of the cell given text to display.
Paint	Yes	Paints the cells of a column.
SetDataGridInColumn	No	Does special processing when the column is added to the DataGrid control.

Listing 1 The Edit method

```
protected override void Edit(CurrencyManager source, int rowNum, Rectangle bounds,
    bool isReadOnly, string instantText, bool cellIsVisible)
{
    if (_inEdit)
        return;

    // The Link column is NOT editable but when in edit mode switches
    // to hyperlink mode
    if (cellIsVisible && !_inAbort)
    {
        ChangeEditMode(true);
        _llControl.SetBounds(bounds.X, bounds.Y, bounds.Width, bounds.Height);
        _llControl.Text = GetDisplayText(source, rowNum);

        // Define the link area (1 covering the whole string)
        _llControl.Links.Clear();
        string url = (string) GetColumnValueAtRow(source, rowNum);
        _llControl.Links.Add(0, _llControl.Text.Length, url);
        _toolTip.SetToolTip(_llControl, url);
        _llControl.LinkClicked +=new
                              LinkLabelLinkClickedEventHandler(_llControl_LinkClicked);
    }
    else
    {
        ChangeEditMode(false);
        _inAbort = false;
    }

return;
}
```

The life of a `DataGrid` column is spent switching between two states—read and edit. When one of the cells is selected (there's always one selected cell in a selected row), the column enters edit mode, and the underlying control gets displayed within the boundaries reserved to the cell on the screen. The control should supply any UI elements that can be used to edit the current value. The column is also responsible for committing changes. The column doesn't have to physically store the new value but must pass it to a built-in method for actual storage. The data is stored in the bound data source object care of the `DataGrid`'s internal framework.

When the user clicks outside the cell, or completes the update procedure, the edit mode is reset, the control is hidden, and the content of the cell is rendered through normal painting. Let's see how all this turns to code with a concrete example—a hyperlink column.

Create a Hyperlink Column

The class I'm going to create is called `DataGridLinkColumn` and inherits from the standard base class—`DataGridColumnStyle`. The class constructor is quite simple and limited to initializing the control behind:

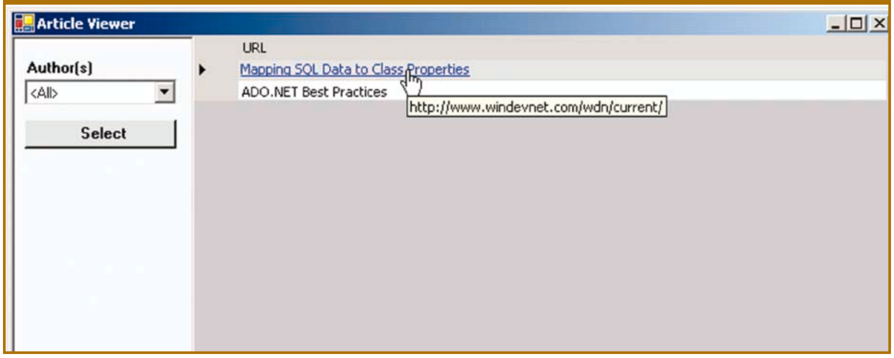
```
public DataGridLinkColumn() : base() {
    Initialize();
}
private void Initialize() {
    _llControl = new LinkLabel();
    _toolTip = new ToolTip();
    _llControl.Visible = false;
    _inEdit = false;
    _inAbort = false;
}
```

The `DataGridLinkColumn` is expected to render a database-stored URL string as a clickable hyperlink. In the .NET Framework, there's just one control that serves that purpose—the `LinkLabel` control. The `LinkLabel` control is a special type of label that allows you to define sensitive areas and associate them with custom information—typically, but not necessarily, a URL. The control renders any link area with blue underlined text, thus mimicking a web hyperlink. However, when one clicks on any hyperlinked areas, an event is fired to the application but no URL navigation takes place by default. This is an important point to keep in mind.

As the previous code snippet suggests, the `LinkLabel` control provides no specific support for tooltips—another thing that was easy to obtain with web hyperlinks. If you want to add tooltip support to the URL displayed through the grid, a second control—a ToolTip control—must be instantiated. The `LinkLabel` control is created and marked as hidden.

Table 1 lists the `DataGridColumnStyle` methods that you might want to override to implement a real-world custom column. The base class `DataGridColumnStyle` is marked abstract and so are some of its methods. It goes without saying that you have to override all abstract methods, otherwise a compile error will catch you. In addition, to all the necessary overrides, a couple of other methods that

Figure 2 The sample application in action



If you want to architect a column optimized for date values, the DateTimePicker control sounds like the perfect choice

have a concrete implementation in the base class should be overridden too—`ConcedeFocus` and `SetDataGridInColumn`.

The `Edit` method is invoked when the user clicks to select (and edit) a particular cell. The `LinkLabel` control is configured to work and bound to the URL and the text for the cell. The `column` class also defines three custom properties aimed at defining the binding mechanism between the column and the grid's data source. The properties are `Text`, `DisplayMember`, and `NavigateUrlField`.

The former two regard the text of the hyperlink; the latter is about the URL to jump to. If you set the `Text` property, then all the cells of the column will display the same text; otherwise, the values in the `DisplayMember` field are used. Likewise, the values in the `NavigateUrlField` are used to link the cells on a per-row basis. Listing 1 shows the source code of the `Edit` method. The `ChangeEditMode` helper function sets internal flags that determine the modality of the column. The `LinkLabel` control is resized to fit into the room of the selected cell. It is also assigned text depending on the `Text` and `DisplayMember` values.

The Links collection of the `LinkLabel` control tracks the sensitive areas of the control. A sensitive area is given by a range of characters in the bound text. A link area is a pair of integers indicating the start position and the width of a clickable substring. The following sample code defines a single link area that spans over the length of the text:

```
_llControl.Links.Add(0, _llControl.Text.Length, url);
```

The third argument—the URL—is not used to navigate upon clicking. That value, instead, is passed on to the event handler that will take care of the click event on the `LinkLabel` control.

```
void _llControl_LinkClicked(object sender,
LinkLabelLinkClickedEventArgs e)
{
    string target = e.Link.LinkData as string;
    if (target != null)
        Process.Start(target);
}
```

Listing 2 Defining the columns to bind to the DataGrid

```
private void ConfigureGrid()
{
    // Define the table style
    DataGridTableStyle skin = new DataGridTableStyle();
    skin.MappingName = "Titles";

    // #1 -- Title column
    DataGridLinkColumn columnTitle = new DataGridLinkColumn();
    columnTitle.MappingName = "URL";
    columnTitle.HeaderText = "Title";
    columnTitle.NavigateField = "URL";
    columnTitle.DisplayMember = "title";
    columnTitle.Width = 190;
    skin.GridColumnStyles.Add(columnTitle);

    // Add the table style info (must occur HERE)
    grid.TableStyles.Add(skin);
}
```


When the `LinkClicked` event fires, the event handler gets a data structure that contains a `Link` field. The `LinkData` property of the field stores the URL (or any other information) that was associated with the area. To complete the demonstration and launch the browser, you call the static `Start` method on the `Process` class. The `Process` class is defined in the `System.Diagnostics` namespace.

If the `Edit` method is at the heart of the column class, the `Commit` method represents another critical point because it is where any changed value is validated and stored into the data source. Since the `DataGridLinkColumn` class doesn't show editable values, the implementation of `Commit` is minimal. (See the companion source of this article and the MSDN documentation for more information.)

Painting the Cell

What happens when a cell is not selected for editing? As mentioned, the underlying control is hidden and the default, read-only, text is painted on the window background. The `DataGridColumnStyle` class defines three overloads for the method `Paint`. One of these overloads holds a concrete implementation; two of them are abstract and must be overridden in any derived class.

The `Paint` method must accomplish one key thing: drawing the text for the cell using the current visual settings such as font and colors. In particular, you have to determine the background and the foreground brushes according to the grid's style for items, alternating items, and selected items. In addition, you must select the grid's font to avoid any graphical discrepancy.

The `Paint` method is also the only way you have to implement more ambitious forms of customization over a `DataGrid` control. For example, if you want to draw cells with negative values in red (or code any similar, data-driven logic) you can do that only by overriding the `Paint` method.

Putting It All Together

Figure 2 shows the sample application in action. At first, the cells in the URL column display as normal text. As soon as you click to select the cell, the text morphs into a `LinkLabel` control, and by clicking, you jump to the desired URL. How can you bind a custom column to a `DataGrid` control?

If you want to have total control over the columns displayed, you have to define a table style. Listing 2 shows how to do that. Each column is represented by a column object added to the `GridColumnStyles` collection of the `DataGridTableStyle` object. The grid table style object is the skin used to build the grid's user interface. The grid can have multiple skins if needed.

Summary

Creating custom columns for a Windows Forms `DataGrid` control is a necessity if you're going to use it in real-world applications. Although powerful, the default version of the control is rarely optimal for a realistic end-user application. So far, resorting to a third-party grid was a foregone conclusion. While that still remains an effective option, the customization layer of the built-in `DataGrid` control can help you roll your own customization and increase your expertise. On the down side, experimentation takes time. What's certain is that the `DataGrid` control is a component more powerful than any other we had in the past bundled with Visual Basic. Evaluating its cost-effectiveness in the economy of a particular project, though, is completely up to you. **w::d**

| Download code › **windevnet.com/wdn/code/** |

GDI+ isn't just for .NET apps

Drawing on GDI+ From Native C++

ONE OF THE BIG selling points of .NET is the new graphics library called GDI+. Reading through all of the articles and books written about GDI+, you get the impression that it is a feature only available to .NET developers, but this, in fact, is far from the truth. The GDI+ library is implemented as a native DLL and the classes in .NET are merely a wrapper around this native code. The GDI+ DLL is provided as part of Windows XP, but it is also installed with the .NET redistributable. This native code is provided through exported DLL functions, and the Platform SDK provides a thin wrapper class library that gives a friendlier face to these functions. In this article, I will describe these wrapper classes and explain how you can write rich graphics with native C++.

Getting Started

The GDI+ library is contained in a DLL called "GdiPlus.dll," and it's provided with XP as part of the operating system. Normally, a system library is located in the %systemroot%\System32 folder because LoadLibrary automatically searches this folder for system libraries, and this is the case when you install the .NET framework on Windows 2000. However, this universal dumping ground approach has caused all kinds of problems in the past, particularly when more than one version of a library is available. XP has native support for multiple versions of libraries through side-by-side installation and manifest files. XP calls such shared libraries "assemblies," but make no mistake—these are not .NET assemblies, they do not contain .NET code; instead, they are native code DLLs that are located and loaded using an extended version of the Windows DLL loader. A manifest file is an XML file that contains versioning information about the library, and an application can also have a manifest file to indicate the version of the libraries that it expects.

When a side-by-side shared library is installed on XP, Windows Installer will store the library in a folder in the side-by-side assembly cache: %systemroot%\WinSxS. The subfolder is named according to the version of the library, so different versions of the same library are available on the machine. For example, my XP machine has service pack 1 installed, which included a new version of GdiPlus.dll, so there are two versions of this DLL with the following file versions as seen through the properties dialog in Windows Explorer:

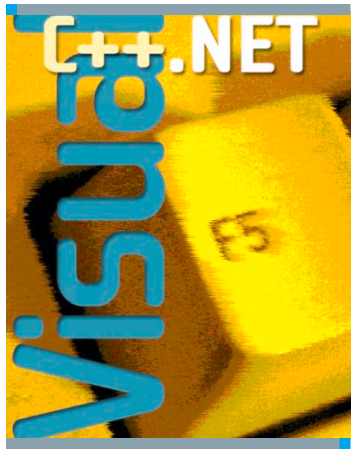
```
5.1.3097.0 (xpclient.010817-1148)
5.1.3101.0 (xpsp1.020828-1920)
```

The first file is GDI+ Version 1.0.0.0, the second file is GDI+ Version 1.0.10.0.

When XP loads an application, it examines the manifest file provided with the application to determine the version of GDI+ that the application requires. If the manifest or the version information is not available, XP will load the latest version of the library from the side-by-side assembly cache.

Headers and Libraries

As with all Windows libraries, you can access the GDI+ facilities through exported functions. If you run DUMPBIN /EXPORTS on this library, you will see over 600 functions with names that start with



Gdip. The GDI+ library is object oriented and it contains various "classes" for things like fonts, bitmaps, and brushes. However, DLLs are not class based; instead, functions are exported as C functions, so each method of each GDI+ "class" is exported as a standalone C function. Of course, a `class` method is applied to a particular instance of the class and has access to the instance through a `this` pointer. This is simulated with the exported functions through the first parameter, which is an opaque pointer to a data structure of the particular object type. The definitions of these functions can be found in the GdiPlusFlat.h header file in the Platform SDK and the data structures are defined in GdiPlusGpStubs.h. The import library is called GdiPlus.lib.

For example, the functions associated with solid brushes are:

```
GpStatus WINGDIPAPI GdipCreateSolidFill
   (ARGB color, GpSolidFill **brush);
GpStatus WINGDIPAPI GdipSetSolidFillColor
   (GpSolidFill *brush, ARGB color);
GpStatus WINGDIPAPI GdipGetSolidFillColor
   (GpSolidFill *brush, ARGB *color);
GpStatus WINGDIPAPI GdipCloneBrush
   (GpBrush *brush, GpBrush **cloneBrush);
GpStatus WINGDIPAPI GdipDeleteBrush
   (GpBrush *brush);
GpStatus WINGDIPAPI GdipGetBrushType
   (GpBrush *brush, GpBrushType *type);
```

The first function can be considered the constructor, which will create a brush with a particular alphaRGB color. This function returns an instance of `GpSolidFill`, which is passed as the first parameter to the other solid fill brush functions. The last three functions in this list can be used with any of the GDI+ brush types and `GdipDeleteBrush` can be considered the destructor of brushes, and is used to clean up the resources used by the brush. If you look up `GpBrush` and `GpSolidFill` in GdiPlusGpStubs.h you'll see these definitions:

```
class GpBrush {};
class GpSolidFill : public GpBrush {};
```

RICHARD GRIMES *is an author and speaker on .NET. His latest book,* Programming with Managed Extensions for Microsoft Visual C++ .NET, *updated for Visual C++.NET 2003, is available now from Microsoft Press. He can be contacted at richard@richardgrimes.com.*

These classes have no members, which reinforces the fact that when they are used as parameters to the `Gdip` functions, they are opaque parameters and do not refer to any data that you can use.

Although you can call these GDI+ exported functions in your application code, the flat API is a pain to use, and you do not have the advantages of object-oriented encapsulation. To address this issue, Microsoft has provided C++ thin wrapper classes that give access to these GDI+ objects through C++ objects. In total, there are 30 header files describing these classes with one master header file, GdiPlus.h, that includes them all. The GdiPlusFlat.h header file is included in GdiPlus.h in a namespace called "DllExports" to isolate the flat API.

A Basic GDI+ Application

GDI+ does not change the basic windowing facilities of Windows, it just gives you a richer set of APIs to draw in an existing window. So, an application that uses GDI+ must create a window (and register a class for the window) and implement a message pump to dispatch messages meant for the window to a custom windows procedure. This is how a Windows application has always been implemented, and there is no change with GDI+. But, the GDI+ library must be initialized before you can use it, and it must be shut down before your process finishes—this is performed by the functions `GdiplusStartup` and `GdiplusShutdown`, respectively.

Listing 1 shows the basic code for initializing the GDI+ library; the initialization occurs before the message pump starts. `GdiplusStartup` is passed a `GdiplusStartupObject` to indicate the version of the library and to allow you to provide a callback function for debug builds. You can also indicate in this parameter if you would prefer to replace the GDI+ background thread with your own thread, in which case the final parameter is used to return `hook` and `unhook` functions to setup and shutdown this thread. Once GDI+ has initialized, you will be returned a token in the first parameter that is passed to `GdiplusShutdown` after the message pump has completed and all GDI+ objects have been released.

Once you have initialized the library, you can use the GDI+ objects in your paint handlers. The process is straightforward and is considerably helped by the wrapper classes that perform clean up of the wrapped GDI+ object in the destructor so that you can allow variables to go out of scope to clean up resources. The workhorse of GDI+ is the `Graphics` class, which should be familiar to .NET developers. An instance of this class is created by passing a device context handle to the constructor, or a pointer to an `Image` object if you want to draw to an offscreen buffer. The `Graphics` class has all of the methods that you need to draw lines, curves, strings, bitmaps, and shapes.

Solid shapes are drawn using a `Brush` object, and lines are drawn using `Pen` objects; pens and solid brushes are constructed from a `Color` object that gives both the RGB value of the color and an Alpha value that determines the transparency of the color. Listing 2 shows example code to draw a gray string in the center of a window's client area. The `DrawString` method takes a `Font` object and a `Brush` object to indicate how the text should be drawn. I have used the version of `DrawString` that takes a bounding rectangle as a parameter, so I also have to pass a `StringFormat` object to indicate how the string will be drawn inside this rectangle: In this case, the string should be centered vertically and horizontally.

GDI+ offers many features that are not part of GDI. I have already mentioned that colors have an Alpha value, which means that you can blend colors by drawing two colors at the same position. Another way to do this is with a gradient brush, where you supply two colors and two points and GDI+ will blend from one color to the other along the line defined by the points. GDI+ also has native support for cardinal splines, and support for multiple image formats such as JPEG, GIF, and TIFF.

Coordinate Systems

GDI+ uses three coordinate systems to rationalize the GDI coordinate system, which are called world, page, and device. The units that you pass to the `Graphics` methods are world units, which can have any scale and any origin. These units are converted to page units, which always have their origin at the top left-hand corner of the client area, but can be scaled to real-world units like inches or millimeters. Finally, page units are converted to device units, which are pixels. You have control over these units through transforms. Page units are set through `Graphics::SetPageUnit()`. For example, to use inches you call:

```
Graphics graphics(hdc);
graphics.SetPageUnit(UnitInch);
```

If you draw a line straight after these statements, each unit will be an inch and the origin will be the top left-hand corner. Further, the x-coordinate increases from left to right and the y-coordinate increases from top to bottom. Since your windows will only be a few inches wide or high, it means that most of the coordinates that you will use will be fractions of an inch. This is why rectangles, points, and sizes are represented with classes that have floating-point numbers. However, the width of the pen that is used will also be in inches, and the default width will be one unit, which will give you a wide pen! It makes more sense to use smaller units. To do

Listing 1 Basic entry point for an application that uses GDI+

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE,
    LPSTR, int iCmdShow)
{
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken,
                   &gdiplusStartupInput, NULL);
    // User method to register a windows class
    RegisterMyClass(hInstance);
    // User method to create a window of that class
    HWND hWnd = CreateMyWindow(hInstance);
    ShowWindow(hWnd, iCmdShow);
    UpdateWindow(hWnd);

    MSG msg;
    while(GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    GdiplusShutdown(gdiplusToken);
    return msg.wParam;
}
```

Listing 2 Sample code to print a string in the center of the window

```
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hwnd, &ps);

    RECT rect;
    GetClientRect(hwnd, &rect);

    Graphics graphics(hdc);
    // Use a gray brush
    SolidBrush brush(Color(255, 128, 128, 128));

    LPCWSTR str = L"Windows Developer Magazine";
    int len = wcslen(str);

    // Calculate the font size to fill the window width,
    // assume average width is less than height of font
    int pixelsPerChar = (rect.right-rect.left)/len;

    FontFamily fontFamily(L"Arial");
    Font font(&fontFamily, pixelsPerChar,
              FontStyleRegular, UnitPixel);

    RectF client(rect.left, rect.top,
                 rect.right, rect.bottom);

    // Indicate that we want the string in the
    // center of the window
    StringFormat format;
    format.SetAlignment(StringAlignmentCenter);
    format.SetLineAlignment(StringAlignmentCenter);

    // Draw the string
    graphics.DrawString(str, -1, &font, client, &format, &brush);
    EndPaint(hwnd, &ps);
}
```

Listing 3 GDI+ code to shift the origin to the center of the window

```
RECT rect;
GetClientRect(hwnd, &rect);
float xOrigin = 0.5f * (rect.right - rect.left)/
    (graphics.GetDpiX() * graphics.GetPageScale());
float yOrigin = 0.5f * (rect.bottom - rect.top)/
    (graphics.GetDpiY() * graphics.GetPageScale());
graphics.TranslateTransform(
    xOrigin, yOrigin, MatrixOrderAppend);
// Draw a 1 inch box with a gradient pen
LinearGradientBrush grad(
    PointF(-50.0f, 0.0f), PointF(50.0f, 0.0f),
    Color(255, 255, 0, 0), Color(255, 0, 0, 255));
// 1 inch square, centered on the origin
graphics.FillRectangle(&grad, -50, -50, 100, 100);
```

this you can pass a scaling factor to `Graphics::SetPageScale()`, and to indicate that each unit is 100th of an inch you can call:

```
graphics.SetPageScale(0.01f);
```

If you want to shift the origin or change the direction of the axes, you have to apply a world transform, which will convert the world units (in the format you prefer) to page units. World transforms can be applied in one of two ways: through calling the various `Transform` methods on the `Graphics` class or by creating a `Matrix` object. To mirror along the x-axis and to get the y-coordinates to increase from bottom to top, the y-coordinate should be scaled by –1:

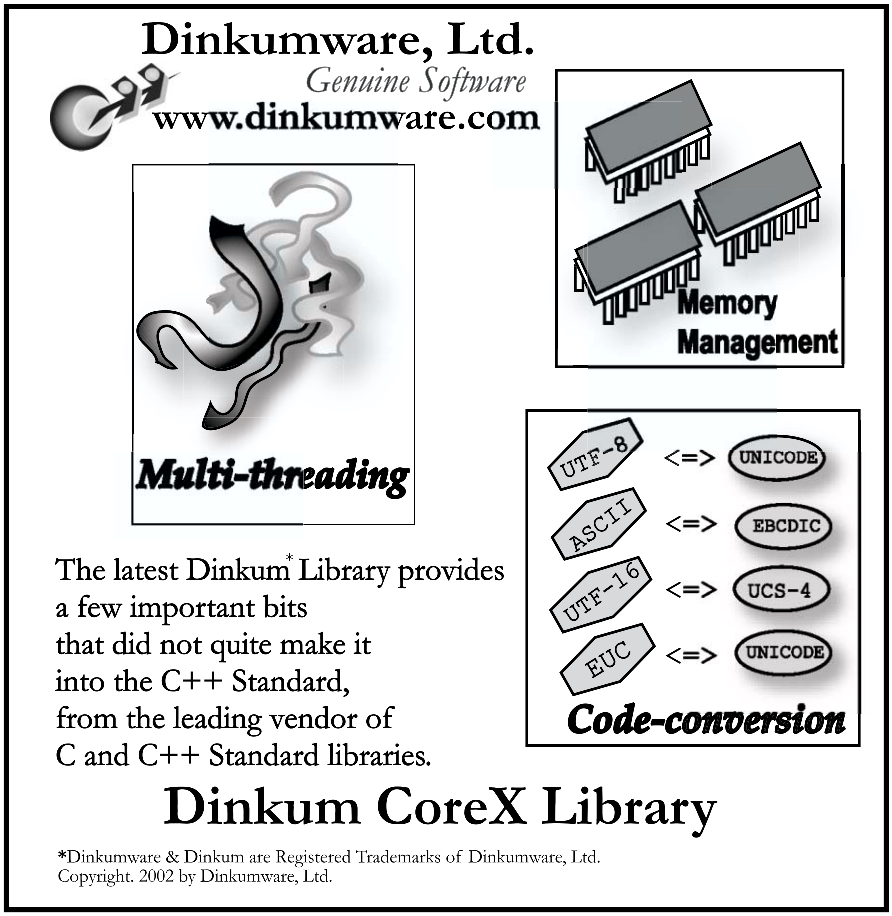
```
graphics.ScaleTransform(1.0f, -1.0f,
               MatrixOrderAppend);
```

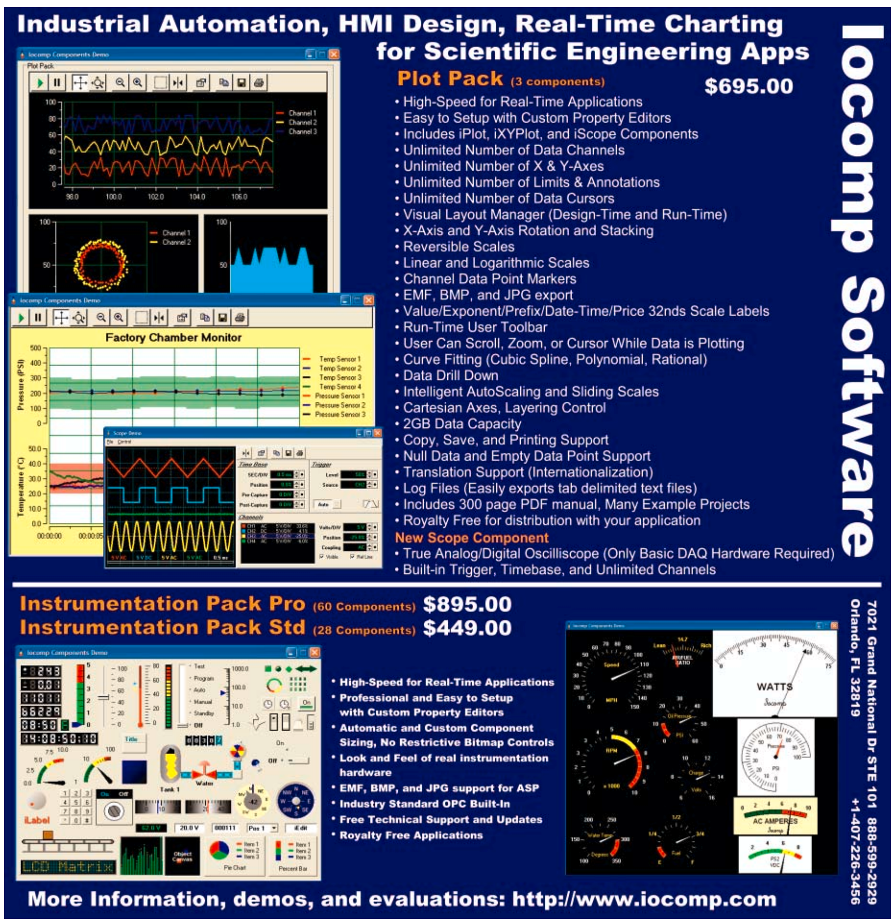
Moving the origin is just as simple: Call the `TranslateTransform` method with suitable values to move the origin from the top left-hand corner to the new location. However, you have to take into account the other transforms that might be applied. So if we assume that the page transforms outlined earlier will be performed on the coordinates, we need to scale the location of the new origin. Listing 3 shows how to place the origin in the center of the window assuming the transforms shown already. First, this code gets the width of the client area in device units and divides by two to get the center of the window in device units. This is then scaled to world units by dividing by the page scale and by the screen resolution in pixels per inch. The y transform is calculated in a similar way.

Wrap Up

GDI+ brings you great new graphic features: graphics transforms, cardinal splines, alpha blending and gradient brushes, and support for creating and rendering multiple image formats. All of these facilities are available to non.NET code because GDI+ itself is a native code library. The GDI+ C++ wrapper classes offer a simplified way to access the GDI+ functions making unmanaged graphics code simple and straightforward to write. **w::d**

| Download code › **windevnet.com/wdn/code/** |





GET ADDITIONAL INFORMATION ABOUT PRODUCTS AND SERVICES YOU SEE ADVERTISED FAST!

PHONE: Contact the vendor directly using the information in the advertisement.

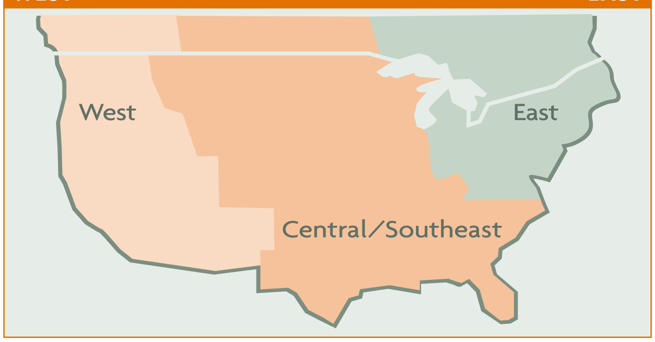
WEB: Go to the development tool page on our web site, www.windevnet.com. From there you can link to the advertisers below.

www.windevnet.com

windows:: developer

APPLICATION DEVELOPMENT FROM WINDOWS TO WEB NETWORK

WEST **EAST**



Michele Hurabiell	Regional Manager—West	
415-947-6199	mhurabiell@cmp.com	
Ed Day	Regional Manager—Central/Southeast	
785-838-7547	eday@cmp.com	
Jon Hampson	Regional Manager—East	
603-924-8500	jhampson@cmp.com	
Julie Thibault	Account Manager—All Regions	
603-924-8400	jthibault@cmp.com	

NEW PRODUCTS

Submit new product announcements to wdletter@cmp.com.

Trolltech Announces Qt Script for Applications

Trolltech has made Qt applications scriptable with the release of Qt Script for Applications (QSA). Written to the Qt API, QSA takes static Qt/C++ applications and makes them dynamic. The QSA toolkit include: the QSA SDK, which allows Qt developers to make their applications scriptable; Qt Script, a multiplatform interpreted scripting language based on the ECMAScript standard; QSA Workbench, a lightweight scripting environment including code formatting, syntax highlighting, code completion, and stack-trace output; and the Input Dialog Framework, a high-level GUI API that allows scripters to write dialogs.

Trolltech
650.813.1676
www.trolltech.com

Universe Online Releases WebControls 2.0

Universe Online has announced WebControls 2.0, a suite of JavaScript web-based controls for all popular browsers and ASP, JSP, ASP.NET, and J2EE platforms. The WebControls 2.0 framework uses an object-oriented approach to application development. The Control suite includes menu, toolbar, treeview, panel, and button controls. Features include flexible menu and toolbar item layouts, collapsible or expandable menu and toolbar items, use of formatted HTML and embedded forms with any item, absolute and relative positioning, custom images with rollovers, and server-side helpers for ASP.NET, Java, and ASP environments. WebControls 2.0 ranges in price from $99.00–$249.00, depending on configuration.

Universe Online Inc.
248.681.1274
www.uolweb.com

TMG Development Ships ASP.NET Localizer

TMG Development's ASP.NET Localizer toolkit aims to provide all the design and run-time support necessary for producing an internationalized web site using ASP.NET. Using ASP.NET Localizer, a designer types the localized strings and image source locations into the Visual Studio designer, and Localizer generates a new .resx file for that language, with all the property changes written to this new file. The developer can switch backwards and forwards between languages using Localizer, and the design surface is kept up to date with the current language selection. When the web application is run, Localizer checks the current locale and generates the web form using resources for that locale.

Localizer 1.0 requires Microsoft Visual Studio .NET, and the Microsoft .NET runtime version 1 or above. An individual license for Localizer costs $199.99.

TMG Development Ltd
www.winformreports.co.uk

Compuware Delivers DriverStudio 3.0

The DriverStudio suite of tools is designed to help device-driver developers write, debug, test, and tune driver code that meets the standards for Windows Hardware Quality Labs (WHQL) driver certification. DriverStudio 3.0 now integrates with Microsoft Visual Studio .NET as well as with Visual Studio 6. Support has been added for USB 2.0 and AVStream drivers, Connection Oriented NDIS drivers, and wireless IEEE 802.11b drivers. Additionally, the IEEE1394 (Firewire) and PLX9056 chipset support has been improved. DriverStudio 3.0 is currently shipping at a U.S. price of $2499.

Compuware
313.227.7300
www.compuware.com

BitShape Launches BitShape iEdit 2.0

BitShape has released BitShape iEdit 2.0, a multifunctional text editor for web programmers and web designers. BitShape iEdit has a fully customizable editor, supports tag insertion from the toolbar, and works directly with PHP/Apache servers. The program also supports bookmarks inside the code. The results of any changes in the code are immediately shown in the program's Internal view window. Other features include templates, text processing and formatting, syntax coloring, different encoding options, smart pasting and indentation, a clipboard viewer, WinXP/Win2K style menus, and support for plug-ins. BitShape iEdit is 2 MB in size and costs under $30.00.

BitShape
www.bitshape.com

Emurasoft Updates EmEditor

EmEditor is a text editor for Windows that fully supports Unicode. The editor features an assortment of plug-ins for specific needs, while maintaining a small and fast core program. EmEditor's features include clickable URLs, search and replace, and keyboard, toolbar, and menu customization. Syntax highlighting is supported for many programming languages. EmEditor is compatible with Windows XP and shares the Windows XP look and feel. A single user license costs $30.00.

Emurasoft Inc.
425.882.9988
www.emeditor.com

Microsoft Unveils Java Language Conversion Assistant 2.0

Built on ArtinSoft migration technology, the Java Language Conversion Assistant (JLCA) automates the process of migrating language syntax and library calls from existing Java language source code into Visual C# .NET. With Version 2.0 of the JLCA, developers can convert JSPs and servlet applications to ASP.NET. Microsoft has also published a JSP to ASP.NET conversion guide, which features a step-by-step code conversion of the http://www.codenotes.com/ web site. The guide supports the documented migration with videos, white papers, sample code, and additional resources. The JCLA 2.0 is available on the MSDN web site at http://msdn.microsoft.com/vstudio/downloads/tools/jlca/.

Microsoft Corporation
425.882.8080
www.microsoft.com

Desktop Software Offers XL Report Builder v2.0

XL Report Builder from Desktop Software allows users to build reports with user-defined parameters from any ODBC database, in Microsoft Excel format. Macros or built-in Visual Basic can be used, and reports can be built singly or in books. Reports can use several sources in different formats; it is possible to generate a single report using MS SQL and Access. A command-line interface is provided as well as the GUI. SQL queries are supported for data set creation, and stored procedures can be called. XL Report Builder v2.0 costs $99.00.

Desktop Software Ltd
www.dswsoft.com

Spying on .NET Applications

Using global hooks, the CLR hosting interface, and the .NET remoting interface to monitor the flow of events in your .NET apps

THE .NET FRAMEWORK HAS an ambitious goal of providing a completely new environment for Windows programming. The .NET library provides a complete replacement for most of the Win32 API, but there are few Win32 API features that have no corresponding classes in the .NET library. Most of these API routines can be accessed by a .NET program using Platform Invoke. One of the most important Win32 features, which cannot be easily accessed from a .NET application, is Windows hook support. Dino Esposito provides a partial solution for this problem in "Windows Hooks in the .NET Framework"(*MSDN Magazine*, October 2002 [1]), but his approach covered only local (thread) hooks, which are running in the same process as the .NET application.

The question of how to implement global hooks in .NET persists. This interest is understandable: For many years, global hooks were used to implement such powerful applications as Spy++, automated testing, accessibility tools, and popup stoppers. Lack of global hook support in .NET explains why Spy++ was not updated in Visual Studio.NET, unlike most other tools and utilities. Spy++ is the primary debugging tool for Windows UI programming because it provides a dynamic view of windows and messages, something a debugger cannot do. Unfortunately, Spy++ became almost useless for .NET programmers. It can only display windows and messages—not .NET objects and events. When developing .NET GUI applications using the Windows Forms library, I felt a need to see a hierarchy of .NET objects, examine their properties, and monitor events.

DMITRI LEMAN *is a consultant in Silicon Valley specializing in .NET and Java development. He can be reached at DmitriL@Forwardlab.com.*

That's why I developed a technique of injecting a .NET object into the address space of another .NET application, which allows me to bring Spy into the new age. In this article, I will explain how I used global hooks, the CLR hosting interface, the .NET remoting interface, and reflection to achieve my goal. Sources for a simple .NET spy are available for download from the *Windows Developer* web site as well as from my web site (http://www.forwardlab.com/). This simple spy displays a list of running .NET applications on a computer, and allows me to select one, inject an agent into it, display a tree of Windows Forms controls in that application, and watch the flow of events. I also developed a more sophisticated DotNetSpy application, which allows me to examine and modify properties on objects, execute methods, and select events to watch. It is also available from my web site. Using the presented technique and sources, it should be easy to develop powerful new .NET tools for debugging, automated testing, system monitoring, and so on.

Design

After studying the problem, I divided it into four parts. The first is how a .NET application (spy) can inject code (agent) into another application space (target). The second is how the agent can attach to an existing instance of the .NET Runtime working inside the target. The third is how to examine .NET objects and monitor events in the target. And the fourth is how to communicate information back to the spy.

While looking for answers for the first question, I studied several techniques, such as the CLR Debugging API, the .NET remoting API, and Windows hooks. The debugging API is very powerful and will meet my needs. It allows, among other things, enumerating running processes, attaching to a process or starting a new one, and injecting a code into the debuggee and executing it. For more information about the debugging API see, "Common Language Runtime Debug Overview," an article on CLR Debugging in *MSDN Magazine* [2]. I decided not to use the debugging API primarily because it will not allow spying on an application while running under Visual Studio debugger (since two debuggers cannot be attached to the same application simultaneously). I used the debugging API only to enumerate the running .NET processes.

.NET remoting is the official way of "accessing objects in other application domains." At first I was fooled by this title and hoped that it would solve all my problems. A quick study of .NET documentation revealed that remoting is based on the client-server model, which requires both applications to run objects that are designed to communicate with each other. This, obviously, will not meet my goal of injecting an agent into any .NET application without raising its suspicion.

Finally, I decided to use a Windows hook to inject the agent. A limitation of the hook approach is its inability to target applications without windows, such as Windows services. This is not a problem for the spy, which is designed to work with GUI applications. If there is a need to target nonGUI applications, other means of cross-application penetration can be used, such as the `CreateRemoteThread` API The hook approach requires writing a native (unmanaged) Win32 DLL, then calling this DLL from a .NET application using Platform Invoke. The local hook solution described in [1] did not require a native DLL because a .NET delegate was passed as a callback to the `CallNextHookEx` routine. Since the local hook callback is called on the same thread, the delegate remains valid and accessible. This, obviously, is not possible in my case since I want the callback to be called in the target application space. Therefore, I should either use a global hook (associated with all threads on the system) or a hook associated with the thread, which owns a window in the target application. The second question is the most challenging—how can Win32 code peek inside a .NET Runtime environment and inject a .NET agent into it? One approach is to use the CLR Debugging API, but we already discarded it.

The second technique is to use the CLR hosting interface. The key routine of this interface is `CorBindToRuntimeEx`. As the name of this function suggests, it does exactly what I need: It lets Win32 code bind to the CLR environment. Unfortunately, the documentation says that hosts use this API to load CLR into a process but does not mention that it can also be used to attach to an already running CLR. Therefore, I will have to use this routine in an undocumented way. There is a danger that in future versions, Microsoft may patch this loophole and break my spy. Then I will have to find a more tricky way to achieve my goal. But in .NET Versions 1.0 and 1.1, `CorBindToRuntimeEx` works perfectly. It provides access to the `IcorRuntimeHost` interface, which lets hosts start and stop CLR, enumerate domains running in the process, and create and configure new domains. The hook code will enumerate existing domains (represented by the `_AppDomain` interface), then call the `CreateInstanceFrom` method on each domain to create an instance of the agent .NET class inside that domain. For a more traditional use of CLR hosting, see [3]. This completes the most complex part of the spy design. At this point, the agent object is instantiated inside the target application and is ready to take it under control.

> Some objects are remotable; others aren't. Nonremotable objects can't be represented in another app domain. Remotable objects can be either marshaled by reference or by value

To explore the structure of objects in the target application, the agent will use methods of the `System.Windows.Forms.Control` class. This is the base class for other classes in the `System.Windows.Forms` assembly, which have corresponding Win32 windows. The .NET Framework maintains a hierarchy of `Control`-based objects, which mostly corresponds to the hierarchy of windows. Therefore, to let the spy display a windows hierarchy, the agent should find the top-level control in the target application and then recursively enumerate its children. The agent will use the `Control.FromHandle` static method to get a `Control` object corresponding to the window handle passed by the spy. Then the `Control.TopLevelControl` property will give the main control and the `Control.Controls` collection will provide access to children. The agent will also bind handlers to several events in each `Control` object, such as `Click`, `GotFocus`, `KeyDown`, `MouseDown`, and others. These handler routines (located in the agent) will report events to the main spy for printing. A more sophisticated spy should allow the user to monitor selected events on selected controls, but the simple spy included with this article will monitor a fixed selection of events on all discovered controls.

Spy should also display properties of objects. This can be done by direct reading properties of the `Control` object, such as `Width`, `Height`, and so on. Alternatively, the agent can use the .NET reflection API to examine all fields and properties (including private) of the `Control` object. Here lies the biggest advantage of .NET Spy over old Spy++. Over the last decade, many new controls were introduced such as tree, list view, and so on, but Spy++ was frozen, displaying only the coordinates, style, and a few other properties of a window. Like a Visual Studio debugger, .NET Spy can display all fields and properties of whatever class the target application uses to implement a window.

The final question is: How can the agent communicate the collected information back to headquarters? Here .NET remoting fits nicely. When designing a communication using a remoting API, it is necessary to assign the roles of client and server, determine the activation mode, choose a channel type, and decide what classes have to be transported, how to transport them (by value or by reference), and how to share metadata. In the case of Spy, I decided to make the main Spy application play the role of a server and make the server object a singleton. One or many agents may be simultaneously active in different target applications. An agent should instantiate a client class, which should connect to the server and get a reference to the remote server object using the `System.Activator.GetObject` method. Then the client should call a method on a server and pass a reference to itself. The server should maintain a collection of all currently running clients (agents). This should allow the spy and the agent to call methods on each other to pass requests, get results, and monitor events. When the target is terminated normally, the destructor of the client should be called and the client should remove its registration from the server. Both client and server classes should belong to the same assembly packaged as a DLL. A path to this DLL should be passed to the `_AppDomain.CreateInstanceFrom` routine discussed earlier.

The .NET remoting interface provides a choice of two channels: HTTPChannel and TCPChannel. HTTPChannel uses HTTP protocol and SOAP format (by default) to transport method calls and objects. HTTPChannel is the best for communication across the Internet and through firewalls. Since Spy and agents run on the same computer, TCPChannel is the best choice because it has less overhead. Each client and the server should register an instance of TCPChannel with two important properties: port and name. Most .NET samples use a fixed port number, but it may cause conflicts with other applications. Therefore, I decided to pass port number `0` to have the system automatically assign an unused port. Then the server should call `GetUrlsForUri` to get the URL (which includes an automatically assigned port) and the client should use that URL to connect to the server. Spy will use the HookDLL to pass the URL to the agent.

The final issue to decide is which objects should be exchanged between the agent and the spy. Some objects are remotable while others are nonremotable. Nonremotable objects cannot be represented in another application domain. Remotable objects can be marshaled either by reference or by value. The objects marshaled by reference are represented in another domain by proxies. A client calls the proxy, which transfers calls to the original object. All modifications made by the client to the state of the object stays with the object. Objects marshaled by value are copied and recreated in the remote domain. All calls and property changes made by the client to these objects only affect the copy and are never propagated to the original object. As I already explained, the server and the agent should exchange references to be able to call each other. This means that both the client and server should be marshaled by reference. This is achieved by deriving these classes from `System.MarshalByRefObject`. Finally, the agent will use several small classes to pass collected information to the server. These classes should be marshaled by value, which is done by marking them with the `[Serializable]` attribute. I considered passing Control and other objects from the target application to the spy by reference to let the spy examine them using reflection. This worked for standard .NET classes, but an exception was thrown if an object from the application's private assembly was passed. This means that unless the spy loads all private assemblies of all target applications, it should not directly touch references to their objects. Therefore, the agent should dump all properties and fields of these objects to a string and pass them to the spy as a string. Now the whole picture of the spy design is clear and shown in Figure 1.

Implementation

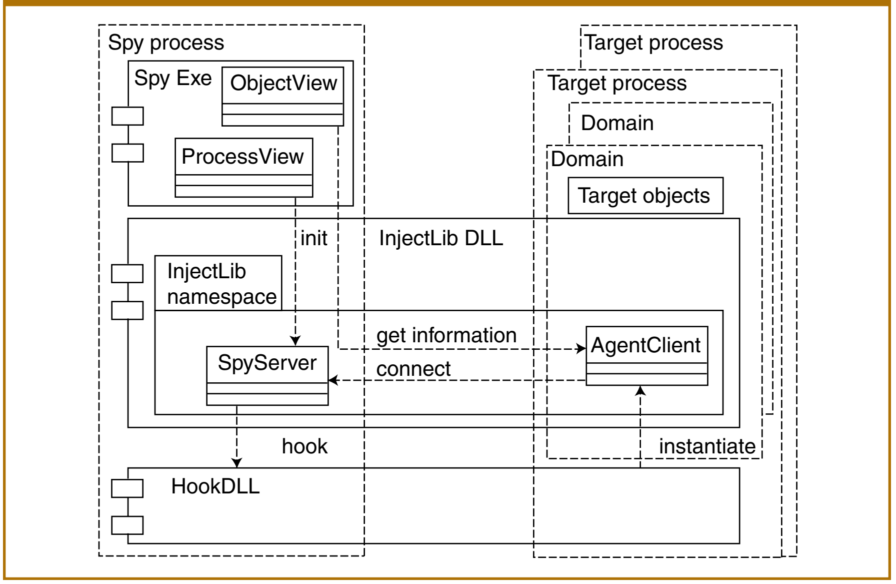
As Figure 1 shows, there are three components in Spy: Spy.exe, InjectLib.dll, and Hook.dll. The first two are .NET-managed components and will be written in the C# language. Hook.dll is an ordinal Win32 DLL and will be written in C++. Visual Studio.NET can manage these different types of projects in the same solution (workspace). Therefore, I first created a blank solution called "SimpleSpy," then added a new Visual C# project using the Windows Applications template and named it "SpyGUI." After that, I added another Visual C# project using the Class Library template and named it "InjectLib." Then I added a new Visual C++ project using the Win32 project template, named it "HookDLL," and selected option "DLL" in the Win32 Application Wizard.

Next, I wrote the Windows hook implementation in the file HookDLL.cpp. It exports a single function, InjectSpyAgent, with four arguments: target window handle, path to InjectLib assembly DLL, agent object name, and server URL. This function copies arguments to a shared memory area, sets the CallWndProc hook for the target window's thread, sends a message to that window, and removes the hook. The CallWndProc hook routine calls the Bind routine (shown in Listing 1), which does the actual injection. It is important to avoid accidental loading of the .NET Framework into applications that don't use it. Therefore, HookDLL should not statically link to CLR libraries and should not call LoadLibrary on any of them. Instead, Bind calls GetModuleHandle("mscoree"). "mscoree" is the DLL that exports the CorBindToRuntimeEx method, which gives access to the main CLR hosting interface ICorRuntimeHost. Then it calls the EnumDomains method on this interface and enumerates domains using the NextDomain method. Each domain is represented by interface AppDomain, which is a native representation of managed interface System._AppDomain. This interface has (among others) several overloaded CreateInstance methods. The native version of the interface has these methods numbered (because the C language does not support overloading). The Bind routine calls the CreateInstanceFrom_3 method to instantiate the agent in the domain. In order to use these interfaces, HookDLL.cpp has the line #import "mscorlib.tlb", which includes a native definition of many .NET interfaces (including _AppDomain). I also included headers "mscoree.h" for the CLR hosting interface definitions and "corhdr.h" for various other .NET definitions. I added the function EnumProcesses to the HookDLL to enumerate .NET processes using the IcorPublishProcessEnum interface from the .NET debugging interface.

To implement the second component, named "InjectLib," I renamed "Class1.cs" (created by Visual Studio) to "InjectLib.cs." This source file will contain the namespace InjectLib with a few classes: SpyServer, AgentClient, MemberDescr, and ClassDescr. SpyServer (shown in Figure 1) is a singleton performing the role of a .NET remoting server. It is instantiated by the Spy GUI and is used to communicate with AgentClient. AgentClient is instantiated inside the target application(s) by the HookDLL. As explained in the design section, SpyServer and AgentClient extend MarshalByRefObject, allowing them to exchange remote references with each other. Two small classes, ClassDescr and MemberDescr, are used to pass information from the agent to the server. These classes are marked by a [Serializable] attribute to be marshaled by value. All their members must be serializable as well. I only used members of type String and ArrayList. At the beginning of InjectLib.cs there are several using statements that permit use of classes in Windows Forms, remoting, diagnostics, collections, and reflection namespaces without prepending full names to all classes. To compile successfully, the project should have references to assemblies containing all used namespaces. Some references were added automatically when the project was created, but I needed to add references to System.Windows.Forms and System.Runtime.Remoting manually using the Project | Add Reference dialog.

At first, Spy GUI calls the static Init method in the SpyServer class. This method creates and registers an instance of the TCPChannel class using ChannelServices.RegisterChannel() and an instance of WellKnownServiceTypeEntry with RemotingConfiguration.RegisterWellKnownServiceType(). This well-known service type entry creates an association of a URI (I use string SimpleSpy) to the class (SpyServer) and the activation pattern (singleton). Then I pass the URI SimpleSpy to GetUrlsForUri on the channel to get the URL such as "tcp://localhost:1031/SimpleSpy." 1031 is an automatically assigned port, and SimpleSpy is the URI associated with the SpyServer class. Now any other process on the same computer may get access to the single instance of SpyServer inside the Spy GUI by calling Activator.GetObject(typeof(SpyServer), URL). The SpyServer.Init method has a few lines dealing with the incompatibilities between .NET Runtime 1.0 and 1.1. To strengthen security, Runtime 1.1 does not allow passing object references through remoting channels unless the TypeFilterLevel property of the serialization provider is set to TypeFilterLevel.Full. This breaks programs written for 1.0 SDK. Programs written for 1.1 will not work under Runtime 1.0 because property TypeFilterLevel does not exist in 1.0. To let the spy work under both run times, I decided to use reflection GetProperty() and SetValue() methods. I also wrote the PrintRemotingConfiguration method, which prints all registered client and service type entries to help in debugging.

Figure 1 The .NET Spy components include Spy.exe, InjectLib.dll, and Hook.dll



Spy GUI needs a reference to the instance of SpyServer, but it cannot simply call new SpyServer() because the remoting framework will create another instance later, which will violate the singleton design. Therefore, the Spy GUI calls AgentClient.Connect()—the same method that will be called by the AgentClient constructor inside the target application. Like SpyServer.Init, AgentClient.Connect also instantiates and registers TCPChannel and then calls Activator.GetObject with the server's URL to get a reference to SpyServer. Since Spy GUI calls Activator.GetObject in the same domain that the server is registered, a direct reference is returned. When other applications call the same Activator.GetObject, they will get a reference to a proxy representing SpyServer.

SpyServer has a RegisterAgent method, which is called from the constructor of AgentClient—and UnregisterAgent—from the AgentClient.Dispose. These methods add and remove agents to/from the hash table m_Agents. Another method, GetAgent(), is called by the Spy GUI to find an agent in m_Agents for the given process ID. Currently, the simple spy is limited to one agent per process and cannot differentiate between domains. The ReportEvent() method is called by agents to report an intercepted event in the target. Finally, the AgentClient.Dispose() method disconnects all agents, so target applications will not hang after the Spy GUI terminates.

Besides the Dispose and Connect methods already mentioned, AgentClient has GetRelatedWindows and GetWindowProps methods. They are called by SpyServer when the GUI needs to retrieve the window tree and properties of individual window. AppendWindow is an internal method that is called recursively to generate the window tree. Another internal method, AddEventHandlers, adds handlers to several events on a given window. Then there are several event handler routines, such as MouseMoveEventHandler. All of them format event arguments into a string and call ReportEvent, which forwards the call to SpyServer.ReportEvent. This concludes the implementation of the InjectLib component.

Listing 1 The Bind routine

```
//Bind routine works in the address space of the target process.
//It uses CLR hosting interfaces to inject an agent into the CLR environment.
enum HookErrors Bind()
{
    TRACE("Bind()\n");
    ICorRuntimeHost *l_pHost = NULL;
    HCORENUM l_hEnum = NULL;
    IUnknown * l_pUnknown = NULL;
    mscorlib::_AppDomain * l_pDomain = NULL;
    enum HookErrors l_eError = eNoErrors;
    try
    {
        LPWSTR l_pszVersion = NULL;
        HINSTANCE l_hMscoree = GetModuleHandle("mscoree");
        if(l_hMscoree == NULL)
        {
            TRACE("GetModuleHandle(mscoree) returned NULL\n");
            return eNoClrRuntime;
        }
        T_CorBindToRuntimeEx * l_pCorBindToRuntimeEx = (T_CorBindToRuntimeEx *)
            GetProcAddress(l_hMscoree, "CorBindToRuntimeEx");
        if(l_pCorBindToRuntimeEx == NULL)
        {
            TRACE("GetProcAddress(CorBindToRuntimeEx) returned NULL\n");
            return eCannotGetAddrCorBindToRuntimeEx;
        }
        HRESULT l_hResult = l_pCorBindToRuntimeEx(l_pszVersion,
                                NULL,
                                NULL,
                                CLSID_CorRuntimeHost,
                                IID_ICorRuntimeHost,
                                (void **)&l_pHost);
        if(FAILED(l_hResult) || !l_pHost)
        {
            TRACE("CorBindToRuntimeEx() failed. Result %d\n", l_hResult);
            return eCannotGetRuntimeHost;
        }
        l_hResult = l_pHost->EnumDomains(&l_hEnum);
        if (SUCCEEDED(l_hResult))
        {
            while(SUCCEEDED(l_hResult =
                l_pHost->NextDomain(l_hEnum, &l_pUnknown)) && l_pUnknown)
            {//l_pUnknown is System.AppDomain interface
                l_pUnknown->QueryInterface(__uuidof(mscorlib::_AppDomain),
                    (void**)&l_pDomain);

                if(l_pDomain)
                {
                    _bstr_t l_Path(g_szAssemblyPath);
                    _bstr_t l_Name(g_szAgentClass);

                    SAFEARRAY * l_pArray;
                    SAFEARRAYBOUND l_Bounds[1];
                    l_Bounds[0].lLbound = 0;
                    l_Bounds[0].cElements = 1;
                    l_pArray = SafeArrayCreate(VT_VARIANT, 1, l_Bounds);
                    if(l_pArray == NULL)
                    {
                        TRACE("SafeArrayCreate failed\n");
                        l_eError = eSafeArrayFailure;
                    }
                    else
                    {
                        long l_Index;
                        _variant_t l_Element(g_szServerURL);
                        l_Index = 0;
                        l_hResult = SafeArrayPutElement
                            (l_pArray, &l_Index, &l_Element);
                        if(FAILED(l_hResult))
                        {
                            TRACE("SafeArrayPutElement failed. Result %d\n",
                                l_hResult);
                            l_eError = eSafeArrayFailure;
                        }
                        else
                        {
                            TRACE("Before CreateInstanceFrom_3\n");
                            mscorlib::_ObjectHandlePtr l_Handle =
                                l_pDomain->CreateInstanceFrom_3(l_Path,
                                  l_Name,
                                    VARIANT_TRUE,
                                    mscorlib::BindingFlags_Default,
                                    NULL/*Binder*/,
                                    l_pArray, NULL/*culture*/,
                                    NULL/*activationAttributes*/,
                                    NULL/*securityAttributes*/);
                            TRACE("CreateInstanceFrom_3 ret %x\n", l_Handle);
                            l_eError = l_Handle?
                                eNoErrors : eCreateInstanceFailed;
                        }
                        SafeArrayDestroy(l_pArray);
                    }
                    l_pDomain->Release();
                    l_pDomain = NULL;
                }//if(l_pDomain)
                l_pUnknown->Release();
                l_pUnknown = NULL;
            }//while enum
            l_pHost->CloseEnum(l_hEnum);
            l_hEnum = NULL;
        }//if (SUCCEEDED(l_hResult))
        else
        {
            TRACE("EnumDomains() failed. Result %d\n", l_hResult);
            l_eError = eEnumDomainsFailed;
        }
    }
    catch(...)
    {
        TRACE("Exception in Bind()\n");
        l_eError = eException;
    }
    try
    {
        if(l_pHost)
        {
            if(l_pDomain)
                l_pDomain->Release();
            if(l_pUnknown)
                l_pUnknown->Release();
            if(l_hEnum)
                l_pHost->CloseEnum(l_hEnum);
            l_pHost->Release();
        }
    }
    catch(...)
    {
    }
    TRACE("Bind() returns %d\n", l_eError);
    return l_eError;
}//Bind
```

The next step is to implement the Spy GUI. This will be a very simple GUI with two windows classes. The first class (`ProcessesView`) will be a form with a list view and three buttons: `Close`, `Hook`, and `Refresh`. The list view will be populated with running processes on the computer. The second class (`ObjectView`) will also be a form with a label and tree view. This view will be used to display a window's hierarchy and properties for a selected window. Events observed in the target application will simply be printed to the console along with debug information. I used the Visual Studio form designer to define the GUI layout, modify properties, and assign button-click handlers.

Next, I wrote the routine `RefreshProcList`, which calls `System.Diagnostics.Process.GetProcesses()` to get an array of all processes on the current computer. Unfortunately, I didn't find any way from the managed code to determine whether a process has a .NET environment. Therefore, I added a call to the `EnumProcesses` function in HookDLL to enumerate .NET processes. Then I added code to add .NET processes to the list view. The `RefreshProcList` routine is called from constructor and from click handled for the `Refresh` button. It appears that the `Process` object returned from `GetProcesses()` does not hold any resources and, therefore, calling `Dispose()` is not necessary. Then I added the `InitServer` method to perform `SpyServer` initialization. This routine is executed on a separate thread started by the constructor of `ProcessesView`. A handler for the `Hook` button first gets a selected list view item and a `Process` object associated with it. Then it calls the `InjectSpyAgent` method from HookDLL, `SpyServer.GetAgent` to get a remote reference to the agent in the target process. After that it calls `GetRelatedWindows` on this agent and, finally, passes the returned window tree to a new `ObjectView` window object. To compile the call to the unmanaged `InjectSpyAgent` function, it is necessary to write a prototype at the beginning of the `ProcessesView` class. I also assigned `OnObjectViewAction` as a handler for the `ActionEvent` in the `ObjectView`. This method (called when the user double clicks on a window in the tree) calls `AgentClient.GetWindowProps` and creates another instance of `ObjectView` to display the properties of the selected window. `ObjectView` implementation is simple. It has two overloaded `SetInfo` methods to populate the tree view with either window tree or object properties.

Debugging

Spy has code running in different processes— some code is managed, some native. The operation of installing a hook, injecting an agent, and establishing a connection was the most difficult to troubleshoot. It may have been possible to use a combination of Win32 and managed code debuggers, but I decided to use simple print statements to the console. I wrote a simple test application, which contains a single form with many different controls. I converted the Spy GUI and Test applications from Windows Application to Console Application using the `Output Type` property on the projects. Then I wrote a batch file, TestRun.bat, which uses the "start /b" command to launch Test.exe and SpyGUI.exe from the same console. I injected a lot of `Console.Out.WriteLine` calls into most managed methods in the spy and `printf` in HookDLL. I also ensured that all exceptions are caught and printed.

Security

At first glance, the technique presented here is a security breach because it injects an agent into an unsuspecting application and takes it under control (calling methods, modifying fields, and so on). Closer examination shows that the spy does not present a higher risk than any other Win32 application. Any Win32 application can install hooks, create remote threads, and use Win32 and .NET debugging APIs to break into other applications (some limitations may be imposed by NT security). The .NET environment has more sophisticated security, which allows the customization of access to specific resources. By default, code originating from the local computer receives the Full Trust permission set, which gives it access to all resources. The spy always operates on the local computer because Windows hooks cannot be installed remotely. Therefore, by default, the `AgentClient` can access objects, fields, and properties (including private) in the target applications. There are ways to customize .NET security policy on different levels (enterprise, computer, user, and domain). Therefore, it is possible to ban a specific assembly (for example, SpyGUI or InjectLib) from accessing certain resources (for example, unmanaged code). But it is not feasible to explicitly specify all assemblies, which may use the injection technique. It is also possible to programmatically customize .NET security. For example, attributes may be added to an assembly, class, or a member to restrict access. While testing the spy, I found that the `ShowParams` property in the `System.Windows.Forms.Control` class has an attribute that bans access by anybody except for the `System.Windows.Forms` assembly. When `AgentClient` tried to retrieve this property, an exception was thrown. The current version of the simple spy displays this exception instead of the property value (don't be surprised to see a security exception on the console while running the spy). Therefore, my conclusion is that the spy plays by Win32 and .NET security rules and does not pose an additional security risk. Of course, downloading unknown applications from the Internet and running them locally is very dangerous.

Like a Visual Studio debugger, .NET Spy can display all fields and properties of whatever class the target app uses to implement a window

Conclusion

The injection technique presented here uses a combination of a Windows global hook, .NET CLR hosting, remoting, and reflection to build a key to unlock the door to the .NET Runtime environment. It can be used to build a .NET version of Spy++ and automated testing tools. It will be interesting to see what other new tools and applications can be developed using the injection. Unfortunately, a danger remains that the `CorBindToRuntimeEx` function will be modified in the future versions of .NET to close the existing backdoor. Until that happens, we have a good opportunity to debug and test our .NET applications using .NET Spy and other new tools.

References

1. "Windows Hooks in the .NET Framework," Dino Esposito. *MSDN Magazine*, October 2002.
2. "CLR Debugging: Improve Your Understanding of .NET Internals by Building a Debugger for Managed Code," Mike Pellegrino. *MSDN Magazine*, November 2002.
3. "Implement a Custom Common Language Runtime Host for Your Managed App," Steven Pratschner. *MSDN Magazine*, March 2001.

w::d

| Download code › **windevnet.com/wdn/code/** |