CMP
United Business Media

windows developer

APPLICATION DEVELOPMENT FROM WINDOWS TO WEB NETWORK

SPECIAL ISSUE

Windows SECURITY

Aladdin
SECURING THE GLOBAL VILLAGE
eAladdin.com

Protecting your software and business revenue is simply a matter of choosing the right solution.
HASP4® gives you tougher, more reliable software protection than any hardware key on the market. With HASP4, you get:

- Hardware-based string encryption—the strongest way to secure your software.
- A true cross-platform solution: 1 key for 1 source code for Windows®, Mac OS® and Linux®.
- 99.97% hardware success** in the field, backed by ISO 9001:2000 certification.
- A time-based licensing solution with a real-time internal clock—ideal for controlled beta testing, subscription, rental, pay-per-use or any time-based need.
- The widest range of licensing, module and networking models available.
- 24/7 hassle-free remote license upgrades and advanced HASP reporting tools.

Plus, HASP4 is so easy to use, you'll wonder why you didn't choose it before.

Open your eyes to real anti-piracy protection. Call 1-800-562-2543 or visit .com to request your FREE personal HASP4 Developer's Kit today.

HASP®
PROFESSIONAL SOFTWARE PROTECTION

North America: 1-800-562-2543, 847-818-3800 or HASP.us@eAladdin.com **International:** +972-3-636-2222 or HASP.il@eAladdin.com
Germany: HASP.de@eAladdin.com **UK:** HASP.uk@eAladdin.com **France:** HASP.fr@eAladdin.com **Benelux:** HASP.nl@eAladdin.com

* Business Software Alliance Global Software Piracy Study, June 2003. ** Aladdin Knowledge Systems actual hardware key statistics: 1985-2002

windows::developer
APPLICATION DEVELOPMENT FROM WINDOWS TO WEB NETWORK

PUBLISHER Kerry Gates
EDITOR IN CHIEF John Dorsey

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, $64 elsewhere. Payments must be made in U.S. dollars. Make checks payable to *Windows Developer Network*.

CUSTOMER SERVICE: For subscription orders and address changes, contact *Windows Developer Network*, P.O. Box 2092, Skokie, IL 60076 USA. Telephone 800-365-1425 or 847-763-9640; fax 847-763-9606; e-mail wdnetwork@halldata.com. For information on reprints and permissions contact Karen Jacobs at 800-682-4972 ext. 7030.

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 1543-6454) is published monthly by CMP Media LLC, 600 Harrison St., San Francisco, CA 94107 USA, 415-947-6000. Canadian publication registered for GST as CMP Media LLC, GST No. R13288078, Customer No. 2116057, Agreement No. 40011901.

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

FROM THE EDITOR

SECURITY BREACHES IN WINDOWS have made major headlines in the past few months. While the recent strains of the Blaster and soBig e-mail worms may have been new, the message for sys admins and developers is an old one: Any networked system has the potential for being compromised. To help you stay on top of the state of security in Windows development, we've collected articles from some of CMP Media's top technology publications:

"Testing for Software Security," from *Dr. Dobb's Journal*, describes several test-case attacks you can use to uncover unexpected vulnerabiliaties. It also presents a runtime fault-injection DLL that you can use to monitor calls to files, memory, or the registry.

From *MSDN Magazine*, "Defend Your Code with Top Ten Security Tips Every Developer Must Know" gives you the short list of pitfalls for C++, ASP.NET, IIS, SQL Server, and more.

"Implementing an SSL/TLS-Enabled Client/Server on Windows Using GSS API," from *C/C++ User's Journal* uses Microsoft's Security Support Provider Interface API to build an HTTP server with SSL/TLS capabilities.

From *Windows Developer Network*, "Win32 Security in Managed C++," provides techniques for simplifying coding with the Win32 security APIs. It also presents an AccessToken class in Managed C++ and a Security Explorer app for monitoring privileges.

We're also providing two samples of the Windevnet Security Newsletter, "Analyzing the Mescaline Worm" and "Backdoors Can Damage Trust." To stay up to date on Windows security, subscribe to the Windevnet Security Newsletter at http://windevnet.com/newsletters/

John Dorsey
Editor in Chief
wdeditor@cmp.com

CONTENTS

FEATURES

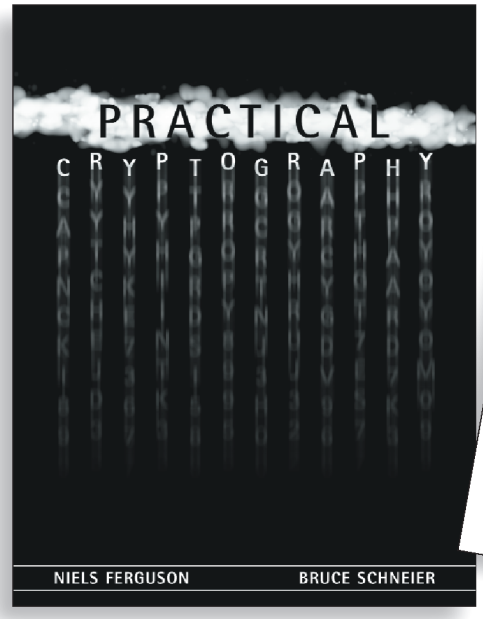
ONLINE EXTRAS

Download these sample chapters from Wiley Publishing Inc.:

- *Practical Cryptography*, by Niels Ferguson and Bruce Schneier, Wiley Publishing Inc., 2003
  Chapter 11 "Primes" and Chapter 12 "Diffie-Hellman"
- *Hiding in Plain Site*, by Eric Cole, Wiley Publishing Inc., 2003
  Chapter 6 "Nuts and Bolts of Steganography"

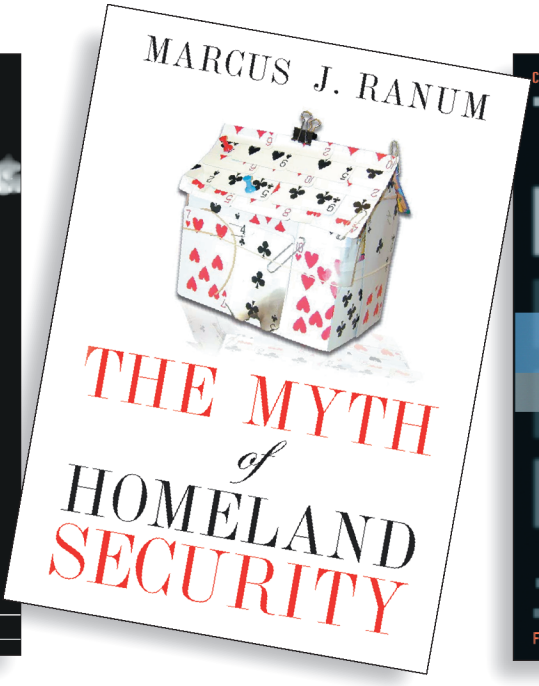
| Download code › **windevnet.com/wdn/code/** |

From computer hacking to security slacking.



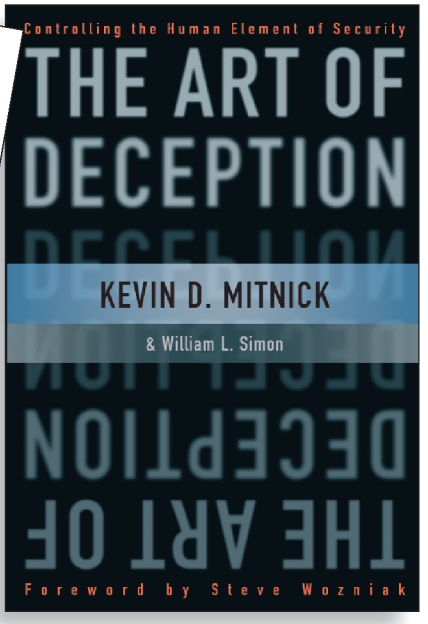
Written by two of the world's top experts in cryptography, this is the long-awaited follow-up guide to the bestselling *Applied Cryptography*. With this first hands-on cryptographic product implementation guide, Niels Ferguson and Bruce Schneier bridge the gap between cryptographic theory and real-world cryptographic applications.

0-471-22357-3



This dose of reality offers specific examples from the Homeland Security Act and explains how they are dangerously bad ideas—but then takes the extra step and provides specific solutions about what can be done to make homeland security a *reality*, as opposed to a myth.

0-471-45879-1



The international bestseller, now in paperback!

The world's most celebrated computer hacker provides specific, detailed guidelines for developing a corporate security manual that addresses the human element of security. He's got the inside scoop, and advises anyone involved with information security protection and policies on how "social engineering" attacks are executed and how they can be prevented.

0-7645-4280-X

WILEY
Now you know.
wiley.com

Testing for Software Security

Rethinking security bugs

SECURITY BUGS ARE DIFFERENT from other types of faults in software. Traditional nonsecurity bugs are usually specification violations; the software was supposed to do something that it didn't do. Security bugs, however, typically manifest themselves as additional behavior—something extra the software does that was not originally intended. This can make security-related vulnerabilities particularly hard to find because they are often masked by software doing what it was supposed to.

Traditional testing techniques, therefore, are not well equipped to find these kinds of bugs. Why? For one thing, testers are trained to look for missing or incorrect output; they see only the correct behavior and neglect to look for other side-effect behaviors that may not be desirable.

For instance, the circle on the left in Figure 1 represents the specification—what the software is supposed to do. The circle on the right represents the true functionality of the application—what the software actually does. Developers and testers are painfully aware that these circles never completely overlap. The area on the left represents either incorrect behavior (the software was supposed to do A but did B instead) or missing behavior (the software was supposed to do A and B but did only A). Traditional software testing is well equipped to detect these types of bugs. Security bugs, however, do not fit well into this model. They tend to manifest as side effects; for instance, the software was supposed to do A, and it did, but in the course of doing A, it does B as well. Imagine a media player that flawlessly plays any form of digital audio or video, but manages to do so by writing the files out to unencrypted temporary storage. This is a side effect that software pirates would be happy to exploit.

It is important that as you verify functionality, you also monitor for side effects and their impact on the security of your application. The problem is that these side effects can be subtle and hidden from view. They could manifest as file writes or registry entries, or even more obscurely as a few extra network packets that contain unencrypted, supposedly secure data.

Luckily, there are both commercially and freely available tools—such as Mutek's AppSight (http://www.identify.com/products/appsightsuite.html) and Holodeck Lite (http://se.fit.edu/holodeck/), respectively—that let you monitor these hidden actions. Another option is to write your own customized monitoring solution such as injecting a custom DLL into the running application's process space.

Creating a Plan of Attack

Software takes input from many different sources. Users, operating-system kernels, other applications, and filesystems all supply input to applications. You have control over these interfaces, and by carefully orchestrating attacks through them, you can uncover many vulnerabilities in the software. Figure 2 is a simple model of software and its interaction with the environment. This model gives you a way to conceptualize these interactions. The four principal classes of input in Figure 2 are:

- Human interface (UI). Implemented as a set of APIs that get input from the keyboard, mouse, and other devices. Security concerns from this interface include unauthorized access, privilege escalation, and sabotage.
- Filesystem. Provides data stored in either binary or text format. Often, the filesystem is trusted to store information such as passwords and sensitive data. You must be able to test the way in which this data is stored, retrieved, encrypted, and managed for security.
- API. Operating systems, libraries, and other applications supply inputs and data in the return values of API calls. Most applications rely heavily on other software and operating-system resources to perform their required functions. Thus, your application is only as secure as the other software it uses and how well equipped it is at handling bad data through these interfaces.
- Operating-system kernel. Provides memory, file pointers, and services such as time

HERBERT THOMPSON *is Director of Security Technology for System Integrity LLC (http://www.sisecure.com).* **JAMES WHITTAKER** *is a professor of computer science at the Florida Institute of Technology. Herbert and James are coauthors of* How to Break Software Security *(Addison-Wesley). They can be contacted at hught@sisecure.com and jw@cs.fit.edu, respectively.* *This article was first published in* Dr. Dobb's Journal, *November 2002.*

and date functions. Any information that an application uses must pass through memory at one time or another. Information that passes through memory in an encrypted form is generally safe, but if it is decrypted and stored even momentarily in memory, then it is at risk of being read by hackers. Encryption keys, CD keys, passwords, and other sensitive information must eventually be used in an unencrypted form and its exposure in memory needs to be protected. Another concern with respect to the operating system is stress testing for low memory and other faulty operating conditions that may cause an application to crash. An application's tolerance to environmental stress can prevent denial of service and also situations in which the application may crash before it completes some important task (like encrypting passwords). Once an application crashes, it can no longer be responsible for the state of stored data. If that data is sensitive, then security may be compromised.

At first glance, it seems as if you could organize a plan of attack by looking at each method of input delivery individually, and then bombard that interface with input. For security bugs, though, most revealing attacks require you to apply inputs through multiple interfaces. With this in mind, we scoured bug databases, incident reports, advisories, and the like, identifying two broad categories of attacks that can be used to expose vulnerabilities—dependency attacks and design-and-implementation attacks.

Attacking Dependencies

Applications rely heavily on their environment to work properly. They depend on the OS to provide resources such as memory and disk space, the filesystem to read and write data, the registry to store and retrieve information, and on and on. These resources all provide input to the software—not as overtly as human users do, but input nonetheless. Like any input, if the software receives a value outside of its expected range, it can fail.

When failures in the environment occur, error-handling code in the software (if it exists) gets called. Error handlers tend to be the weak point of an application in terms of security. One reason for this is that failures in the software's environment that exercise these code paths are difficult to produce in a test lab situation. Consequently, tests that involve disk errors, memory failures, and network problems are usually only superficially explored. It is during these periods that the software is at its most vulnerable and where carefully conceived security measures break down. If such situations are ignored and other tests pass, we are left with a dangerous illusion of security. Servers do run out of disk space, network connectivity is sometimes intermittent, and file permissions can be improperly set. Such conditions cannot be ignored as part of an overall testing strategy. What's needed is a way to integrate these failures into your tests so that you can evaluate their impact on the security of the product itself and its stored data.

Creating environmental failure scenarios can be difficult, usually requiring you to tamper with the application code to simulate specific failing responses from the operating system or some other resource. This approach isn't very feasible in the real world, however, because of the amount of time, effort, and expertise it takes to simulate just one failure in the environment. Even if you did decide to use this approach, the problem is determining where in the code the application uses these resources and how to make the appropriate changes to simulate a real failure in the environment.

One alternative approach is run-time fault injection: Simulating errors to the application in a black-box fashion at run time. This approach is nonintrusive and lets you test production binaries, not just contrived versions of your applications that have return values hard coded. There are several ways to do this; in the example presented

Listing 1

```
#include "stdafx.h"
#include <windows.h>
typedef HMODULE (WINAPI *loadlibrary_t) (LPCWSTR, HANDLE, DWORD);
loadlibrary_t real_LoadLibraryExW;
DWORD dwAddr;
/* Our imposter function for the real LoadLibraryExW. All it does is check
if the incoming filename is msrating.dll and either returns NULL and
sets an appropriate error, or lets the call go through to our saved header
instructions of the real function which then jump to the real function
in the appropriate location.
*/
HMODULE WINAPI imposter_LoadLibraryExW(LPCWSTR lpFileName,
                                                    HANDLE hFile, DWORD dwFlags)
{
    if (!_wcsicmp(lpFileName, L"msrating.dll"))
    {
        SetLastError(ERROR_FILE_NOT_FOUND);
        return NULL;
    }
    else
    {
        return real_LoadLibraryExW(lpFileName, hFile, dwFlags);
    }
}
BOOL APIENTRY DllMain( HANDLE hModule, DWORD  ul_reason_for_call,
                                                        LPVOID lpReserved
                      )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
      // Allocate memory for copying the first few instructions of the target
      // function. Since the granularity on VirtuallAlloc is a page, might as
      // well allocate 4096 bytes
      real_LoadLibraryExW = (loadlibrary_t) VirtualAlloc(NULL, 4096,
                                        MEM_COMMIT,PAGE_EXECUTE_READWRITE);
     // Copy first two instructions of LoadLibraryExW (which we know add up
     // to 7 bytes - we need 5 for our jump).
     memcpy((void *) real_LoadLibraryExW, (void *)LoadLibraryExW, 7);

     // Writes a jump instruction out right after the copied instructions.
     // The jump is a relative near jump to the 8th byte of LoadLibraryExW.
     PBYTE pbCode = (PBYTE) real_LoadLibraryExW + 7;

     // Write opcode for jump near and move (write) pointer forward
     *(pbCode++) = 0xe9;

     // Write out address of where to jump to using a double word pointer.
     // That way, compiler takes care to put it in big endian convention.
     PDWORD pvdwAddr = (PDWORD) pbCode;

     // Write out address - the +3 = -4 +7 (for the offset into the function)
     *pvdwAddr = (DWORD) LoadLibraryExW - (DWORD) pbCode + 3;

     // Move (write) pointer forward the length of the address.
     pbCode+=4;
     DWORD dwOld, dwTemp;

     // Set the page with LoadLibraryExW to writeable
     VirtualProtect((LPVOID) LoadLibraryExW, 4096,
                                                PAGE_EXECUTE_READWRITE, &dwOld);
     // Write out the jump
     pbCode = (PBYTE) LoadLibraryExW;

     // Write opcode for jump near to the beginning to LoadLibraryExW
     *((PBYTE) LoadLibraryExW) = 0xe9;

     // Compiler gymnastics to move forward by *1* byte and not 4 to get
     // the exact address where to write the target address for the jump to.
     pvdwAddr = (PDWORD) (pbCode + 1);
     dwAddr = (DWORD) pvdwAddr;

     // Write the address
     *pvdwAddr = (DWORD) imposter_LoadLibraryExW - (DWORD) LoadLibraryExW - 5;

     // Set the old protection back. This is very important for some Win32
     // functions. They refuse to work with writeable protection enabled.
     VirtualProtect((LPVOID) LoadLibraryExW, 4096, dwOld, &dwTemp);

     break;
   }
 return TRUE;
}
```

here, we overwrite the first few bytes of the actual function to be called in the process space and insert a JMP statement to our fault injection code in its place. There are other methods that can be used as well, such as modifying the import address tables; a technique for which we have found Jeffrey Richter's *Programming Applications for Microsoft Windows*, Fourth Edition (Microsoft Press, 1999) to be an excellent reference.

Using these techniques, you can redirect a particular system call to your own impostor function. One passive use for this is to simply log events. This can be informative for the security tester because it lets you watch the application for file, memory, and registry activity.

At this point, you are in control of the application and can either forward a system request to the actual OS function or deny the request by returning any error message you choose. This technique is illustrated in the first attack.

Block access to libraries. Applications rely on external software libraries to get work done. Operating-system libraries and third-party DLLs are critical for the application to function properly. As testers and developers, it is your responsibility to ensure that failures here do not compromise the security of your application. By denying a library to load, you have deprived the application of some functionality it expected to use. If the application does not react to this failure by displaying an error message, this may be a sign that appropriate checks are not in place and that the software may be unaware that this code did not load. If the library in question provides security service, then all bets are off.

You can deny a library to load in Windows by intercepting the `LoadLibraryExW` function. For instance, consider a publicized bug with Internet Explorer's Content Advisor feature (see "Exposing Software Security Using Runtime Fault Injection" in Proceedings of the *ICSE Workshop on Software Quality*, 2002). If you turn the feature on, all web sites that don't have a RASCi rating are blocked by default. (The Recreational Software Advisory Council, RASCi, rating is assigned to a web site based on its content. This rating system was replaced in 1999, however, with the Internet Content Rating Association, ICRA, rating system.) Listing 1 is the C++ source code of a DLL you can inject into the application to hook the function `LoadLibraryExW` for Windows XP. Our DLL overwrites the first few bytes of this function in the process space of the application under test. These bytes are replaced with a JMP statement to the memory address of our imposter function, `imposter_LoadLibraryExW`.

The problem with IE's Content Advisor is that if IE fails to load the library msrating.dll, users can surf the Web unrestricted. Our imposter function checks to see whether the library that the application is attempting to load is msrating.dll; if so, it blocks the library from being loaded by returning NULL (indicating failure) to the application.

You can uncover clues to library dependencies such as this by changing the code in the imposter function, either to alert you when a specific call is made or log all such calls and their parameters to a file. It then takes a little detective work to determine which services the library is providing to the application and when they are used. With a few modifications to the imposter function, you can then determine what would happen if that functionality were to be denied. Listing 2 is the source of the executable used to inject our DLL into the target application's process space.

In addition to `LoadLibraryExW`, this code can easily be modified to intercept other system calls and monitor and/or selectively deny them at run time. We have developed a freeware tool called "Holodeck Lite" (available electronically at http://se.fit.edu/holodeck/.), using techniques similar to those in Listing One, to help you easily monitor and obstruct common system calls.

Manipulate registry values (Windows specific). The problem with the registry is trust. When developers read information from the

Figure 1 Intended versus implemented software behavior

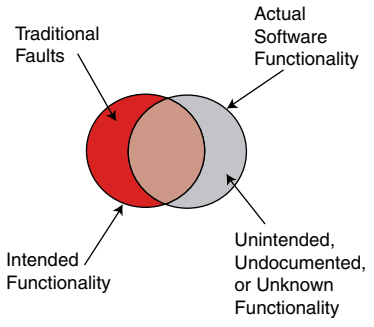
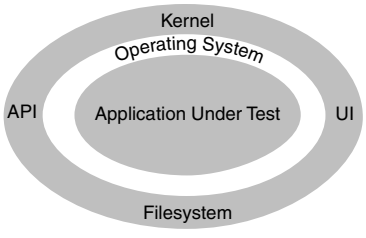


Figure 2 A look at software's users



Listing 2

```
#include "stdafx.h"
#include <windows.h>

/* This program uses one of the simplest injection techniques out there. It
utilizes the fact that parameters and calling convention for LoadLibrary are
the same as the thread function that is suplied to CreateThread/
CreateRemoteThread. It uses that API to call LoadLibrary in the target
process and load the desired DLL.
*/
int main(int argc, char* argv[])
{
    DWORD dwTemp;
    LPVOID pvDllName;

    if (argc < 3)
    {
        printf("Usage: inject commandline dllname.dll\n");
        return 0;
    }

    // Setup the required structures and start the process
    PROCESS_INFORMATION pi = {0};
    STARTUPINFO si = {0}; si.cb = sizeof(si);
    if (!CreateProcess(NULL, argv[1], NULL, NULL, false, NULL,
                                                    NULL, NULL, &si, &pi))
        goto error;

    // Allocate memory for the name of the DLL to be loaded
    if (!(pvDllName = VirtualAllocEx(pi.hProcess, NULL, strlen(argv[2]),
                                          MEM_COMMIT, PAGE_EXECUTE_READWRITE)))
        goto error;

    // Write out the name of the target DLL
    if (!WriteProcessMemory(pi.hProcess, pvDllName, argv[2],
                                           strlen(argv[2]), &dwTemp))
        goto error;

    // Technically this will execute LoadLibrary in the target process with
    // name of the DLL as the first parameter. This relies on the fact that
    // that kernel32.dll will NOT be relocated. Assuming that it won't be, then
    // then address of LoadLibraryA in the target process is the same as ours
    if (!CreateRemoteThread(pi.hProcess, NULL, NULL, (LPTHREAD_START_ROUTINE)
    LoadLibraryA, pvDllName, NULL, &dwTemp))
        goto error;
    return 0;
error:
    if (pi.hProcess)
        TerminateProcess(pi.hProcess, 0);
    printf("Error in injection!\n");
    return -1;
}
```

registry, they trust that the values are accurate and haven't been tampered with maliciously. This is especially true if their code wrote those values to the registry in the first place. One of the most extreme vulnerabilities is when sensitive data, such as passwords, is stored unprotected in the registry.

More complex information can cause problems too. Take, for example, "try and buy" software, where users have either limited functionality or a time limit in which to try the software, or both. In these cases, the application can then be unlocked if it is purchased or registered. In many cases, the check an application makes to see if users have purchased it or not is to read a registry key at startup. We've found that in some of the best cases, this key is protected with weak encryption; in some of the worst, it's a simple text value: 1 purchased; 0 trial.

Force the application to use corrupt/protected files and file names. A large application may read from, and write to, hundreds of files in the process of carrying out its tasks. It's the tester's job to make sure that applications can handle bad data gracefully, without exposing sensitive information or allowing unsafe behavior.

This attack is carried out by taking a file that the application uses and changing it in some way the software may not have anticipated. For a file that contains a series of numerical data that the software reads, for instance, you may want to use a text editor and include letters and special characters. If successful, this attack usually results in denial of service either by crashing the application or by bringing down the entire system. More creative changes may force the application to expose data during a crash that users would not normally have access to.

Force the application to operate in low-memory/diskspace/network availability conditions. Depriving applications of these resources lets testers understand how robust their application is under stress. The decision of which faults to try and when can only be determined on a case-by-case basis. A general rule of thumb, though, is to block a resource when an application seems most in need of it. For memory, this may be during some intense computation the application is doing. For disk errors, look for file writes/reads by the application, then start pounding it with faults. These faults can be simulated relatively easily by modifying the code in Listing One to intercept other system functions, such as `CreateFile`.

Attacking Design and Implementation

It's difficult to identify all the subtle security implications of choices made during the design phase. Looking at a 200-page specification and asking "Is it secure?" will be met with blank looks, even by the most experienced developers. Even if the design is secure, the choices made by the development team during implementation can have a major impact on the security of the product. Here we present some attacks that have been effective at exposing these types of bugs.

Force all error messages. This attack serves two purposes. The first is to see how robust the application is by trying values that should result in error messages and see how many are handled properly, improperly, or not at all. The second is to make sure that error messages do not reveal unintended information to a would-be intruder; for example, during authentication, having one error message that appears when an incorrect user name is entered and having a different error appear when a valid user name is entered but with an incorrect password. At this point, the attacker then knows that they have a correct user name, which means there is now only one string value to attack—the password.

Seek out unprotected test APIs. Complex, large-scale applications are often difficult to test effectively by relying on the APIs extended for normal users alone. Sometimes there are multiple builds a day, each of which has to go through some suite of verification tests. To meet this demand, many applications include hooks that are used by custom test harnesses. These hooks and corresponding test APIs often bypass normal security checks done by the application for the sake of ease of use and efficiency. They are added for testers by developers with the intention of removing them before the software is released. The problem, though, is that these test APIs become so integrated into the code and the testing process that when the time comes for the software to be released, managers are reluctant to remove them for fear of destabilizing the code. It is critical to find these hooks and ensure that if they were to make it out into the field, they could not be used to open up vulnerabilities in the application.

One alternative approach is runtime fault injection

Overflow input buffers. The first thing that comes to many peoples' minds when they hear the term "software security" is the dreaded buffer overflow. For this reason, it is important to test an application's ability to handle long strings in input fields. This attack is especially effective when long strings are entered into fields that have an assumed, but often not enforced, length such as ZIP codes and state names.

API calls have been notorious for unconstrained inputs. As opposed to a GUI where you can filter inputs as they are entered, API parameters must be dealt with internally and checks must be done to ensure that values are appropriate before they are used. The most vulnerable APIs tend to be those that are seldom used or support legacy functionality.

Connect to all ports. Sometimes applications open custom ports on machines to connect with remote servers. Reasons for this vary from creating maintenance channels to automatic updates or possibly as a relic from test automation. There are many documented cases (see http://www.ntbugtraq.com/) where these ports are left open and unsecured. It is important that the same scrutiny that's been given to the communications through the standard ports (Telnet, ftp, and so on) be given to these application-specific ports and the data that flows through them.

Conclusion

Software security testing must go beyond traditional testing if we ever hope to release secure code with confidence. In this article, we have discussed a fault model that describes a paradigm shift from traditional bugs to security vulnerabilities, and outlined some of the attacks testers can use to better expose vulnerabilities before release. These attacks are only part of a complete security-testing methodology. Research into security vulnerabilities, their symptoms, and habits has only just begun.

Acknowledgments

Thanks to Rahul Chaturvedi for providing code excerpts from Holodeck and to Attila Ondi, Ibrahim El-Far, and Scott Chase for their input on this article. w::d

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

Aladdin
SECURING THE GLOBAL VILLAGE
eAladdin.com

Protecting your software and business revenue is simply a matter of choosing the right solution.
HASP4® gives you tougher, more reliable software protection than any hardware key on the market. With HASP4, you get:

- Hardware-based string encryption—the strongest way to secure your software.
- A true cross-platform solution: 1 key for 1 source code for Windows®, Mac OS® and Linux®.
- 99.97% hardware success** in the field, backed by ISO 9001:2000 certification.
- A time-based licensing solution with a real-time internal clock—ideal for controlled beta testing, subscription, rental, pay-per-use or any time-based need.
- The widest range of licensing, module and networking models available.
- 24/7 hassle-free remote license upgrades and advanced HASP reporting tools.

Plus, HASP4 is so easy to use, you'll wonder why you didn't choose it before.

Open your eyes to real anti-piracy protection. Call 1-800-562-2543 or visit .com to request your FREE personal HASP4 Developer's Kit today.

HASP®
PROFESSIONAL SOFTWARE PROTECTION

North America: 1-800-562-2543, 847-818-3800 or HASP.us@eAladdin.com **International:** +972-3-636-2222 or HASP.il@eAladdin.com
Germany: HASP.de@eAladdin.com **UK:** HASP.uk@eAladdin.com **France:** HASP.fr@eAladdin.com **Benelux:** HASP.nl@eAladdin.com

* Business Software Alliance Global Software Piracy Study, June 2003. ** Aladdin Knowledge Systems actual hardware key statistics: 1985-2002

Top Ten Security Tips

Defend your code with top ten security tips every developer must know

This article assumes you're familiar with C++, C#, and SQL

Level of Difficulty: *1* **2** *3*

SECURITY IS A MULTIDIMENSIONAL issue. Security risks can come from anywhere. You could write bad error handling code or be too generous with permissions. You could forget what services are running on your server. You could accept all user input. And the list goes on. To give you a head start on protecting your machines, your network, and your code, here are 10 tips to follow for a safer network strategy.

1. Trust User Input at Your Own Peril

Even if you don't read the rest of this article, remember one thing, "don't trust user input."

MICHAEL HOWARD *is a Security Program Manager in the Secure Windows Initiative group at Microsoft. He is the coauthor of* Writing Secure Code *and the author of* Design Secure Web-based Applications for Microsoft Windows 2000*, both published by Microsoft Press.* **KEITH BROWN** *works at DevelopMentor researching, writing, teaching, and promoting an awareness of security among programmers. Keith authored* Programming Windows Security *(Addison-Wesley, 2000) and coauthored* Effective COM*. He is currently working on a .NET security book. Contact him at http://www.develop.com/kbrown. This article first appeared in the September 2002 issue of* MSDN Magazine.

If you always assume that data is well formed and good, then your troubles are about to begin. Most security vulnerabilities revolve around the attacker providing malformed data to the server machine.

Trusting that input is well formed can lead to buffer overruns, cross-site scripting attacks, SQL injection attacks, and more.

Let's look at each of these potential attacks in more detail.

2. Protect Against Buffer Overruns

A buffer overrun occurs when the data provided by the attacker is bigger than what the application expects, and overflows into internal memory space. Buffer overruns are primarily a C/C++ issue. They're a menace, but generally easy to fix. We've seen only two buffer overruns which were not obvious and were hard to fix. The developer did not anticipate externally provided data that was larger than the internal buffer. The overflow causes corruption of other data structures in memory, and this corruption can often lead to the attacker running malicious code. There are also buffer underflows and buffer overruns caused by array indexing mistakes, but they are less common.

Take a look at the following C++ code snippet:

```
void DoSomething(char *cBuffSrc, DWORD
          cbBuffSrc) {
    char cBuffDest[32];
    memcpy(cBuffDest,cBuffSrc,cbBuffSrc);
}
```

What's wrong with it? Actually, there's nothing wrong with this code if cBuffSrc and cbBuffSrc come from a trusted source, such as code that did not trust the data and so validated it to be well formed and of the correct size. However, if the data comes from an untrusted source and has not been validated, then the attacker (the untrusted source) could easily make cBuffSrc larger than cBuffDest, and also set cbBuffSrc to be larger than cBuffDest. When memcpy copies the data into cBuffDest, the return address from DoSomething is clobbered because cBuffDest is next to the return address on the function's stack frame, and the attacker makes the code perform malicious operations.

The way to fix this is to distrust user input and not to believe any data held in cBuffSrc and cbBuffSrc:

```
void DoSomething(char *cBuffSrc, DWORD
          cbBuffSrc) {
    const DWORD cbBuffDest = 32;
    char cBuffDest[cbBuffDest];
#ifdef _DEBUG
    memset(cBuffDest, 0x33, cbBuffSrc);
#endif
    memcpy(cBuffDest, cBuffSrc, min(cbBuffDest,
              cbBuffSrc));
}
```

This function exhibits three properties of a well-written function which mitigates buffer overruns. First, it requires the caller to provide the length of the buffer. Of course, you should not blindly trust this value! Next, in a debug build, the code will probe the buffer to check that it

is indeed large enough to hold the source buffer, and if not, it will probably cause an access violation and throw the code into a debugger. It's surprising how many bugs you can find when doing this. Last, and most important, the call to memcpy is defensive; it copies no more data than the destination buffer can hold.

During the Windows® Security Push at Microsoft, we created a list of safe string handling functions for C programmers. You can check them out at Strsafe.h: Safer String Handling in C, http://msdn.microsoft.com/library/en-us/dnsecure/html/strsafe.asp.

3. Prevent Cross-site Scripting

Cross-site scripting vulnerabilities are Web-specific issues and can compromise a client's data through a flaw in a single Web page. Imagine the following ASP.NET code fragment:

```
<script language=c#>
    Response.Write("Hello, " +Request.QueryString("name"));
</script>
```

How many of you have seen code like this? You may be surprised to learn it's buggy! Normally, a user would access this code using a URL that looks like this:

```
http://explorationair.com/welcome.aspx?name=Michael
```

The C# code assumes that the data is always well formed and contains nothing more than a name. Attackers, however, abuse this code and provide script and HTML as the name. If you typed the following URL:

```
http://northwindtraders.com/welcome.aspx?name=<script>alert('hi!
</script>
```

you'd get a Web page that displays a dialog box, saying "hi!" "So what?" you say. Imagine that the attacker convinces a user to click on a link like this, but the querystring contains some really nasty script and HTML to get your cookie and post it to a site that the attacker owns; the attacker now has your private cookie information or worse.

There are two ways to avoid this. The first is not to trust the input and be strict about what comprises a user's name. For example, you could use regular expressions to check that the name contains only a common subset of characters and is not too big. The following C# code snippet shows the way that you can accomplish this:

```
Regex r = new Regex(@"^[\w]{1,40}$");

if (r.Match(strName).Success) {
    // Cool! The string is ok
} else {
    // Not cool! Invalid string
}
```

This code uses a regular expression to verify that a string contains between 1 and 40 alphanumeric characters and nothing else. This is the only safe way to determine whether a value is correct.

You cannot squeak HTML or script through this regular expression! Don't use a regular expression to look for invalid characters and reject the request if such characters are found because there is always a case that will slip by you.

The second defense is to HTML-encode all input when it is used as output. This will reduce dangerous HTML tags to more secure escape characters. You can escape any strings that might be a problem in ASP.NET with HttpServerUtility.HtmlEncode, or in ASP with Server.HTMLEncode.

4. Don't Require sa Permissions

The last kind of input trust attack we want to discuss is SQL injection. Many developers write code that takes input and uses that input to build SQL queries to communicate with a back-end data store, such as Microsoft® SQL Server™ or Oracle.

Take a look at the following code snippet:

```
void DoQuery(string Id) {
    SqlConnection sql=new SqlConnection(@"data source=localhost;" +
            "user id=sa;password=password;");
    sql.Open();
    sqlstring= "SELECT hasshipped" +
            " FROM shipping WHERE id='" + Id + "'";
    SqlCommand cmd = new SqlCommand(sqlstring,sql);
...
```

This code is seriously flawed for three reasons. First, the connection is made from the Web Service to SQL Server as the system administrator account, sa. You'll see why this is bad, shortly. Second, notice the clever use of "password" as the password for the sa account!

However, the real cause for concern is the string concatenation that builds the SQL statement. If a user enters an ID of 1001, then you get the following SQL statement, which is perfectly valid and well formed.

```
SELECT hasshipped FROM shipping WHERE id = '1001'
```

However, attackers are more creative than this. They would enter an ID of "'1001' DROP table shipping —", which would execute the following query:

```
SELECT hasshipped FROM
shipping WHERE id = '1001'
DROP table shipping – ';
```

changes the way the query works. Not only does the code attempt to determine if something has shipped or not, it goes on to drop (delete) the shipping table! The—operator is a comment operator in SQL and it makes it easier for an attacker to build a valid, yet dangerous, series of SQL statements!

At this point you're probably wondering how any user could delete a table in the SQL Server database. Surely only admins can do a task like that. You're right. But here you're connecting to the database as sa, and sa can do anything it wants to do on a SQL Server database. You should never connect as sa from any application to SQL Server; rather, you should either use Windows Integrated authentication, if appropriate, or connect as a predefined account with appropriately restricted rights.

Fixing the SQL injection issue is easy. Using SQL stored procedures and parameters, the following code shows how to build a query like this—and how to use a regular expression to make sure that the input is valid because our business dictates that a shipping ID can only be numeric and between four and ten digits in length:

```
Regex r = new Regex(@"^\d{4,10}$");
if (!r.Match(Id).Success)
    throw new Exception("Invalid ID");

SqlConnection sqlConn= new SqlConnection(strConn);
string str="sp_HasShipped";
```

```
SqlCommand cmd = new SqlCommand(str,sqlConn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.Add("@ID",Id);
```

Buffer overruns, cross-site scripting, and SQL injection attacks are all examples of trusting input. All these attacks can be mitigated by believing that all input is evil, until proven otherwise.

5. Watch that Crypto Code!

Now let's look at something near and dear to our hearts. I would say that more than 30 percent of the security code we review contains security mistakes. Probably the most common mistake is homegrown encryption code, which is typically quite fragile and easy to break. Never create your own encryption code; you won't get it right. Don't think that just because you've created your own cryptographic algorithm people won't figure it out. Attackers have access to debuggers, and they have both the time and the knowledge to determine exactly how these systems work—and often break them in a matter of hours. Rather, you should use the CryptoAPI for Win32® applications, and the System.Security.Cryptography namespace has a wealth of well-written and well-tested cryptographic algorithms.

6. Reduce Your Attack Profile

If a feature is not required by 90 percent of clients, then it should not be installed by default. Internet Information Services (IIS) 6.0 follows this plan of installation, and you can read about it in Wayne Berry's article, "Innovations in Internet Information Services Let You Tightly Guard Secure Data and Server Processes," in the November 2002 issue of *MSDN Magazine*. The idea behind this installation approach is that where services that you don't use are running, you don't pay attention to them and they can be exploited. If the feature is installed by default, then it should operate under the principle of least privilege. In other words, do not require the app to run with administrative rights if they are not required. Follow this advice as well.

Figure 1 Assembly Permission Requests

```
using System;
using System.Security.Permissions;

// declare the minimum permissions that this assembly requires to
// even be loaded
[assembly: PermissionSetAttribute(
              SecurityAction.RequestMinimum,
              File="min_perm.xml")]

// optional permissions are permissions that you don't always need in
// order to be functional but that some of your features may rely upon
[assembly: PermissionSetAttribute(
              SecurityAction.RequestOptional,
              File="opt_perm.xml")]

// by specifying the permissions you need, the runtime knows not to grant
// you any other permissions, even new ones that are defined after your
// code ships
```

Figure 2 Serializing Permission Requests

```
using System;
using System.Security;
using System.Security.Permissions;

class App {
    static void Main(string[] args) {
        // serializing a permission set
        IPermission a = new EnvironmentPermission(
            EnvironmentPermissionAccess.Read,
            "MyEnvironmentVar");
        IPermission b = new FileDialogPermission(
            FileDialogPermissionAccess.Open);
        PermissionSet ps = new PermissionSet(
            PermissionState.None);
        ps.AddPermission(a);
        ps.AddPermission(b);
        Console.WriteLine(ps.ToXml());
    }
}
```

7. Employ the Principle of Least Privilege

The operating system and the common language runtime (CLR) have a security policy for several reasons. Many people think that the main reason the security policy exists is to prevent users from intentionally doing bad things: accessing files they shouldn't be allowed to see, reconfiguring the network to suit their needs, and other dastardly deeds. While it's certainly true that insider attacks are common and need to be guarded against, there's another reason for keeping this security policy tight. The security policy is there to put walls around code so that intentional or (just as frequently) unintentional actions by users don't wreak havoc on the network. For instance, an attachment downloaded via e-mail and executed on Alice's machine is restricted to only accessing resources that Alice can access. If the attachment contains a Trojan horse, a good security policy will limit the damage it can do.

When you design, build, and deploy server applications, you cannot assume that every request will come from a legitimate user. If a bad guy manages to send you a malformed request that (heaven forbid) causes your code to behave badly, you want every possible wall around your application to limit the damage. Our point is that the reason your company has a security policy isn't just because it doesn't trust you or your code. It's also there to protect against well-intentioned code that's been exploited by outsiders.

The principle of least privilege says that any given privilege should be granted to the least amount of code necessary, for the least amount of time necessary. In other words, at any given time, try to erect as many walls around your code as possible. When something bad happens—as Murphy's Law guarantees it will—you'll be glad these walls were in place. So here are some concrete ideas for running code with the least privilege possible.

Choose a security context for your server code that grants access only to the resources it needs to get its work done. If certain parts of your code require significantly higher privileges, consider factoring the code out and running just that code with the higher privileges. To safely separate code that runs with different operating system credentials, your best bet is to run this code in a separate process that runs in a more privileged security context. This means you'll need interprocess communication such as COM or Microsoft .NET remoting, and you'll need to design the interface to that code to keep round-trips to a minimum.

If you're using the .NET Framework when factoring your code into assemblies, consider the required level of privilege of each piece of code. You may find it easy to isolate code that requires high privilege into separate assemblies that can be granted more permissions, allowing the majority of your assemblies to run with fewer privileges, thus adding more walls around your code. An easy way to restrict the privileges on a particular assembly is via assembly-level permission requests, as shown in Figure 1. Figure 2 shows how to create the XML files used by these permission requests. If you do this, don't forget that you're limiting not only the permissions of your own assembly, but those of any assemblies you call, due to the Code Access Security (CAS) stack walk.

Many people build their applications so that new components can be plugged in after their product has been tested and shipped. It's very difficult to secure these types of applications because there's no way you can test every possible code path for bugs and security holes. If your application is managed, however, there's a nifty feature provided by the CLR that you can use to lock down these extensibility points. By declaring a permission object or a permission set and calling Per-

mitOnly or Deny, you add a marker on your stack that chokes down the permissions granted to any code you call. By doing this before calling to some plug-in, you can restrict what the plug-in can do. For instance, a plug-in that's supposed to do amortization calculations shouldn't need any access to the file system. This is just another example of least privilege, where you can protect yourself ahead of time. Be sure to document these restrictions and be aware that highly privileged plug-ins will be able to get around these restrictions with the Assert statement.

8. Pay Attention to Failure Modes

Admit it. You hate writing error handling code just as much as the next guy. There are so many ways a piece of code can fail; it's just depressing thinking about it. Most programmers, ourselves included, would much rather focus on the normal path of execution. That's where the real work gets done. Let's get that error handling done as quickly and painlessly as possible and move on to the next line of real code.

Sadly, this is not a safe frame of mind. We need to pay much closer attention to failure modes in code. These bits of code are often written with little attention to detail and often go completely untested. When was the last time you made absolutely sure you stepped your debugger through every single line of code in a function, including every single one of those little error handlers?

Untested code often leads to security vulnerabilities. There are three things you can do to help alleviate this problem. First of all, pay just as much attention to those little error handlers as you do your normal code. Think about the state of the system when your error-handling code is executing. Are you leaving the system in a valid and secure state? Second, once you write a function, step your debugger through it several times, ensuring that you hit every error handler. Note that even this technique may not uncover subtle timing errors. You may need to pass bad arguments to your function or adjust the state of the system in some way that causes your error handlers to execute. By taking the time to step through the code, you are slowing yourself down long enough to take at least a second look at the code and the state of the system at the time it runs. We've discovered many flaws in our programming logic by carefully stepping through code in a debugger; it's a proven technique. Use it. Finally, make sure your test suites force your functions to fail. Try to have test suites that exercise every line of code in your function. These can help you discover regression, especially if you automate your tests and run them after every build.

There's one more very important thing to say about failure modes. Be sure that if your code fails, it leaves the system in the most secure state possible. Here's some bad code:

```
bool accessGranted = true; // optimistic!
try {
    // see if we have access to c:\test.txt
    new FileStream(@"c:\test.txt",
                   FileMode.Open,
                   FileAccess.Read).Close();
}
catch (SecurityException x) {
    // access denied
    accessGranted = false;
}
catch (...) {
    // something else happened
}
```

Let's say that as far as the CLR is concerned, we're granted access to the file. In this case, a SecurityException won't be thrown. But what if, for instance, the discretionary access control list (DACL) on the file doesn't grant us access? In this case, a different type of exception will be thrown. But due to our optimistic assumption in the first line of code, we'll never know this.

A better way to write this code is to be pessimistic:

```
bool accessGranted = false; // pessimistic!
try {
    // see if we have access to c:\test.txt
    new FileStream(@"c:\test.txt",
                   FileMode.Open,
                   FileAccess.Read).Close();
     // if we're still here, we're good!
     accessGranted = true;
}
catch (...) {}
```

This is much more robust, because no matter how we fail, we'll fall back to the most secure mode.

9. Impersonation is Fragile

When writing server applications, you'll often find yourself using, directly or indirectly, a convenient feature of Windows called impersonation. Impersonation allows each thread in a process to run in a distinct security context, typically the client's security context. For instance, when the file system redirector receives a request for a file via the network, it authenticates the remote client, checks to see that the client's request doesn't violate the DACL on the share, then attaches the client's token to the thread handling the request, thus impersonating the client. This thread can then access the local file system on the server using the security context of the client. This is convenient since the local file system is already secure; it will do an access check that considers the type of access being requested, the DACL on the file, and the impersonation token on the thread. If the access check fails, the local file system reports this to the file system redirector, who then can send a fault back to the remote client. This is incredibly convenient for the file system redirector because it simply passes the buck to the local file system and lets the local file system do its own access checking, just as if the client was local.

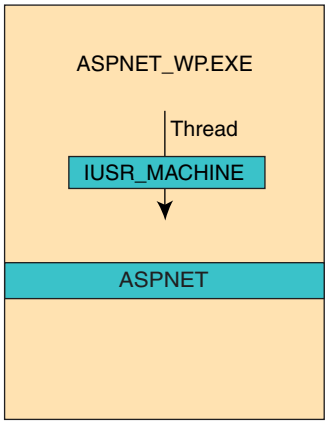
This is all well and good for simple gateways like the file system redirector. However, impersonation is often used in other, more complex applications. Take a Web application for instance. If you're writing a classic unmanaged ASP application, ISAPI extension, or an ASP.NET application which specifies

```
<identity impersonate='true'>
```

in its Web.config file, you are running in an environment with two different security contexts: you have a process token and a thread token, and generally speaking, the thread token will be used for access checks (see Figure 3). Say you are writing an ISAPI application that runs inside the Web server process. Your thread token is likely IUSR_MACHINE, given that most requests are unauthenticated. But your process token is SYSTEM! Say your code is compromised by a bad guy via a buffer overflow exploit. Do you think the bad guy will be content with running as IUSR_MACHINE? No way. It's very likely that his attack code will call RevertToSelf to remove the impersonation token, hoping to elevate his privilege level. In this case, he'll succeed quite nicely. Another thing he can do is call CreateProcess. The token for that new process will be copied not from the impersonation token, but from the process token, so the new process runs as SYSTEM.

What's the solution to this little problem? Well, besides making sure you don't have any buffer overflows to begin with, remember

Figure 3 Checking



the principle of least privilege. If your code doesn't need the godlike privileges afforded to SYSTEM, don't configure your Web application to run inside the Web server process. If you simply configure your Web application to run with medium or high isolation, your process token will be IWAM_MACHINE. You'll have virtually no privileges at all, and this sort of attack won't be nearly as effective. Note that in IIS 6.0, which will be a component of Windows .NET Server, no user-written code runs as SYSTEM by default. This is based on the realization that developers do make mistakes, and any assistance the Web server can provide to reduce the privileges given to code is a good thing, just in case there is a security bug in the code.

Here's another gotcha that COM programmers can run into. COM has a nasty tendency to play games with threads. If you make a call to an in-process COM server whose threading model doesn't match that of the calling thread, COM will execute the call on a different thread. COM will not propagate the impersonation token on the caller's thread, so the result is that the call will execute in the security context of the process, not of the calling thread. What a Surprise!

Here's another scenario where impersonation can bite you. Say you have a server that accepts requests via named pipes, DCOM, or RPC. You authenticate your clients and impersonate them, opening kernel objects on their behalf while impersonating. Let's say you forget to close one of these objects (for instance, a file) when the client disconnects. When the next client comes along, you authenticate and impersonate that client, and guess what? You can still access the file that was "leaked" from the previous client, even if the new client isn't granted access to the file. For performance reasons, the kernel only performs access checks on objects when you first open them. Even if your security context changes later on because you're impersonating somebody else, you will still be able to access this file.

Each of the scenarios we've mentioned so far is a reminder that impersonation is a convenience for server developers, and it's a fragile convenience at that. Pay close attention to your code when you're running with an impersonation token.

10. Write Apps that Non-admins Can Actually Use

This really is a corollary of the principal of least privilege. If programmers continue to produce code that doesn't run well on Windows unless the user is an administrator, how the heck can we ever expect to shake free of the stigma of targeting an "insecure" system? Windows has a very robust set of security features, but if users are forced to run as administrators to get anything done, they aren't getting much benefit from these features.

How can you help? Well first of all, eat your own dogfood. Quit running as an administrator yourself. You will learn very quickly the pain of using programs that were not designed with security in mind. The other day, I (Keith) installed some software provided by the maker of my handheld device that was designed to synchronize data between my desktop and the device. So just as I usually do, I logged off my normal user account, logged back in using the built-in administrator account, installed the software, then logged back to my normal account and tried to run the software. Well, the application promptly popped up a dialog saying it could not access some data file it needed, then proceeded to blow up with an access violation. Folks, this was a piece of software from a major vendor in the handheld space. There is no excuse for this!

After running FILEMON from http://sysinternals.com, I quickly discovered that the application was trying to open up a data file for write access that it had installed in the same directory as its executables. When applications are installed into the Program Files directory like they should be, they should never, ever, try to write data to that directory. There's a reason that Program Files has a restricted access control policy. We don't want users writing to those directories because that could easily allow one user to leave a Trojan horse behind for another user to execute. In fact, this stipulation is part of the basic logo requirements for Windows XP (see http://www.microsoft.com/winlogo/default.mspx).

We hear way too many programmers give excuses for why they choose to run as administrators when developing code. If we all keep ignoring the problem, it's only going to get worse. Folks, it doesn't take admin privileges to edit a text file. It doesn't take admin privileges to compile or debug a program that you started. When you need admin privileges, run individual programs with elevated privileges using the RunAs feature of the operating system (see the November 2001 Security Briefs column). If you are writing tools for developers to use, you have an extra responsibility to the community. We need to stop this vicious circle of folks writing code that only administrators can run, and the only way it's going to happen is if we do it at the grassroots level.

Check out Keith's Web site for more info on how developers can easily run as non-admins at http://www.develop.com/kbrown. Also be sure to pick up a copy of Michael's book, Writing Secure Code (Microsoft Press, 2001), which has tips on how to write apps that run well in a nonadmin environment.

Related Articles

SQL Server Security Modes, http://msdn.microsoft.com/library/en-us/vsentpro/html/veconsqlserversecuritymodes.asp

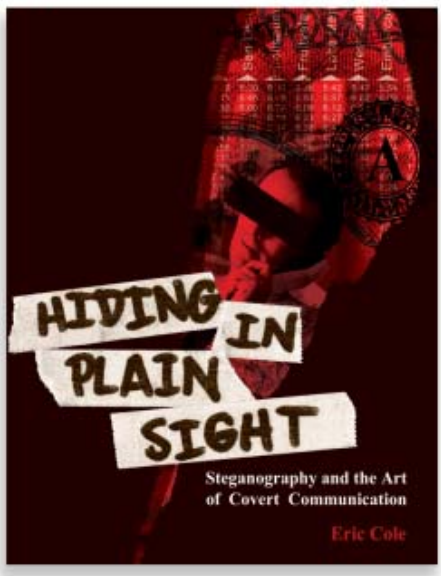
Avoiding Buffer Overruns, http://msdn.microsoft.com/library/en-us/security/Security/avoiding_buffer_overruns.aspw::d

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

Sample Chapters

WILEY

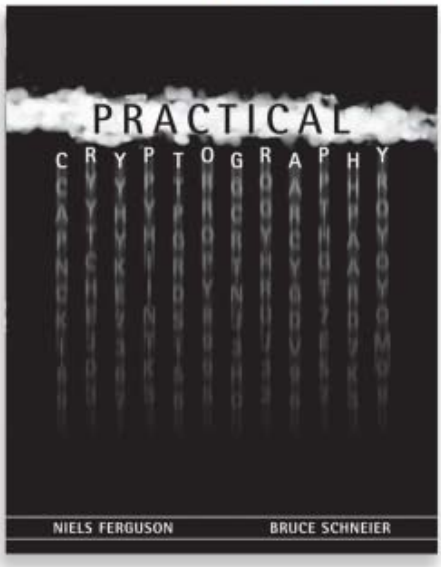
Hiding in Plain Site: Steganography and the Art of Covert Communication by Eric Cole



Your hands-on guide to understanding, detecting, and using today's most potent tool for secret communication--steganography.

These days, encryption of confidential data and communications is an increasingly important part of doing business. But steganography can take data confidentiality to a whole new level, since it hides encrypted messages in ordinary-looking data files, making the very existence of the messages practically undetectable. Although steganography is not a new field and has played a critical part in secret communication throughout history, few people understand exactly how it works today. This detailed, practical guide changes that--whether your goal is to add an extra level of security to business or government communications or to detect and counter steganography when it's used by criminals or terrorists.

Practical Cryptography by David Mayhew and Bruce Schneier



Two of the world's top experts in cryptography teach you how to secure your digital future.

In today's world, security is a top concern for businesses worldwide. Without a secure computer system, you don't make money, you don't expand, and--bottom line--you don't survive. Cryptography holds great promise as the technology to provide security in cyberspace. Amazingly enough, no literature exists about how to implement cryptography and how to incorporate it into real-world systems. With Practical Cryptography, an author team of international renown provides you with the first hands-on cryptographic product implementation guide, bridging the gap between cryptographic theory and real-world cryptographic applications.

Implementing an SSL/TLS-Enabled Client/Server on Windows Using GSS API

Easy, secure communications for Windows

IN RECENT YEARS, WITH the advent of the Internet, secure network communication has become increasingly important and is now present in almost every aspect of the network data exchange. One of the most widely used protocols in that area is Secure Sockets Layer (SSL)[1], which is being gradually replaced by its successor—Transport Layer Security (TLS)[2]. Even though these protocols were specifically designed for the World Wide Web, their utilization is not limited to just the Web traffic — virtually any type of data sent through any type of network can be secured. Having said that, there are protection goals that define what the secure data means in each case. SSL and TLS provide confidentiality (the data is kept secret from the third party that is not involved in the data exchange), message integrity (the message received is the message sent), and endpoint authentication (one of the communication endpoints or both are verifiably known). If you are working on a project that requires implementation of secure communication channel and that channel must provide the above-mentioned types of data protection, selecting SSL or TLS gives the advantage of a well-documented and well-tested, broadly accepted, and robust protocol.

Fortunately for Windows developers, Microsoft has provided a version of Generic Security Service Application Program Interface (GSS-API) [10]— called Security Support Provider Interface (SSPI) [5]. SSPI greatly simplifies the daunting task of digging into SSL internals. Using SSPI slashes the development time threefold since the API effectively shields you from needing to know the details about SSL [3]. All that is left is implementing the underlying transport functionality that supplies the data and the logic to wrap calls to SSPI. And best of all, needed C-libraries and header files are part of the Platform SDK that can be freely downloaded from the Microsoft web site. This article describes how to implement an SSL/TLS client/server system using SSPI. The material in this article requires the reader's familiarity with cryptography, SSL/TLS specifications, and X.509 security certificates.

SSPI is a set of generic functions that can be used to access a specific security provider. Providers are either installed as part of the operating system (NTLM, Kerberos) or added later (Microsoft RSA SChannel provider, Microsoft Exchange provider, etc.) by installing other software. In this article, I will be working with the RSA SChannel provider [5] to implement the SSL/TLS functionality.

SChannel is the Microsoft implementation of the GSS API that wraps the SSL/TLS protocol. In the following paragraphs I'll try to explain in detail how to use SChannel and how it works, but first I'd like to stress a few points about advantages of utilizing SChannel:

Doing so will free the developer from implementing every aspect of the SSL protocol: the gory details are shielded from the developer by the SSPI.

No extra setup is required to run the final application: SChannel is an integral part of the operating system (slight "tweaking" is needed for Win9x family but who runs servers on these systems anyway?). See the next paragraph for the system requirements.

SChannel calls follow GSS API standards. There are, of course, some alternatives—OpenSSL for example. This package is a complete and thorough implementation of the protocol and for someone all too familiar with UNIX is undoubtedly the best choice. The package originally targeted the UNIX community and to compile it relies on the Perl runtime, so some learning curve is required for Windows developers who never worked with UNIX-type systems.

On Windows ME/2000/XP platforms, SChannel is installed and configured by default and does not need additional tuning. For Windows 95/98 OSs a few extra setting up steps are required to ensure that the application you are developing will run smoothly. First, W2K Active Directory client must be installed. The client, `dsclient.exe`, can be found on the Windows 2000 server CD in the Clients\Win9x directory. After the client is installed, a registry entry must be added to the following key `HKLM\System\CurrentControlSet\ Control\SecurityProviders\ SecurityProviders`. Simply append the string:

```
, schannel.dll
```

to the existing `REG_SZ` value. Windows NT machines will require at least Option Pack 4.0.

Provided freely with the article is a project demonstrating implementation of a HTTP server with SSL/TLS capabilities. This server utilizes SSPI wrapper classes—`CSslProvider`, `CSslCredentials`, `CSslTransport`, and `CSspiLib`. Listings 1, 2, and 3 respectively, show partial definition and implementation files for the first three classes. The design goal here was to simplify access to the functionality provided by SChannel. Implementing the handshake or the data exchange phase of the secure protocol with the SSPI can be tedious because of the number of error codes returned and the interpretation of input/output buffer types used.

ALEN TALIBOV *holds an M.S. in electrical engineering from Marine Technical University in St.Petersburg, Russia. He currently works as a software consultant in the metropolitan Boston area. He can be reached atalen@nebutech.com.* *This article was first published in C/C++ Users Journal, August 2003.*

By utilizing the wrappers, your application eliminates the need to keep track of the data flow inside the secure channel, thus you can concentrate on providing the plain or encrypted data. In contrast to the SSPI functions, the wrapper classes have two types of return codes—success (`S_OK`) and failure. This greatly simplifies the decision-making process if any type of error occurs; receiving a failure code should lead to the socket closure, or whatever the transport you are using to receive or send the data.

To deliver the secure context and the encrypted or plain data to the SChannel, I'm providing the `CSslTransport` class. This class can only be obtained through a call to the `CSslProvider`'s function `ObtainTransport`, which ensures that the transport object during its construction receives a reference to the `CSslCredentials` object. The latter in turn provides security credentials during the negotiation phase of the SSL protocol. The sequence of getting the wrappers to do the work required is as follows:

1. Your application calls `CSspiLib::Load()` to prepare the `PSecurityFunctionTable` (see the topic below).
2. Next, the application calls one of the overloaded `Init()` functions of `CSslProvider` to obtain security credentials, used later in the negotiation phase.
3. Once the credentials are obtained, any number of the `CSslTransport` instances can be spawned by calling the `ObtainTransport()` function. When the application is done using the transport object, it should call its `Destroy()` method (see the `CAsyncXferSocket` class for details).
4. Shutting down the application will require calling `CSslProvider`'s `Destroy()` function to release the credentials obtained earlier.
5. The last step is releasing the hold on the SSPI DLL by calling `CSspiLib::Unload()`.

In the case of the demo HTTP server, a descendant of the MFC's `CAsyncSocket` class safeguards the transport object. The SSL transport class fires events to control the inbound and outbound data flow. There are three types of events—`Inbound Data Available`, `Outbound Data Available`, and `Disconnect`. See the `CAsyncXferSocket` class for implementation details and for an example how this class utilizes the new unified event model provided in Visual C++ v7.0. Because of the aforementioned events, to compile and run the project, you'll need Visual C++ v7.0, which is available as part of the VS.NET suite. If you don't have the new compiler, you'll have to modify the code to provide old-fashion callback notification functions similar to these three events fired by the `CSslTransport` object. Note also that the wrapper classes are independent of the MFC and can be plugged into any environment.

SChannel at Work

Before describing in detail the functionality wrapped by the classes mentioned above, I would encourage those readers who are novices to the topic to study the SSL/TLS specifications. Recently a lot of good books on the issue have appeared on the bookstore shelves. If you are serious about developing an SSL-aware application, a book would be a good investment [8], [9].

The first step in taking advantage of the SChannel functionality is loading the SSPI dynamic link library (DLL) and setting up a special table, `PSecurityFunctionTable`, with valid SSPI function pointers. The DLL's name differs depending on the operating system. Windows NT systems have `security.dll`, exposing the SSPI functionality; all other Microsoft systems use `secur32.dll`.

You can download the function table by calling a little nifty procedure, which the SSPI DLL exports as `InitSecurityInterface`. The export is available in two flavors: ANSI and UNICODE, with the letters A and W at the end, respectively, identifying the flavor. To simplify, you may consider using a `SECURITY_ENTRYPOINT` string defined in `sspi.h`, which will map to the correct entry point name in either ANSI or UNICODE environments. The `CSspiLib` class, provided in the supplemental code accompanying this article, wraps DLL initialization and function calls to SChannel using a set of static functions with the same names and parameters that are defined by the SSPI. This class takes care of loading the correct DLL and initializing the function table.

Before SChannel can assist in creating a secure channel, it must tag us with the Windows-type handle: its credentials. This handle is then passed around to other functions, so SChannel knows who is requesting the service and thus does not get confused in servicing multiple users. The handle will also be associated with any special attributes that the principal initiating a session possesses, an X.509 certificate for instance. (Microsoft prefers to use the term 'principal' to identify the entity on behalf of which calls to SChannel functions are made. I'll be using the same term.) For obtaining the handle SSPI provides the function named `AcquireCredentialsHandle()`. A lot of references to structures, function, and error codes mentioned in this article can be found in [5] under the SChannel section.

Multiple overrides of the `CSslProvider` `Init()` function in the sample code provide

Listing 1 CSslProvider class

```
// other code not shown

// definition

class CSslProvider
{
public:
  CSslProvider(void);
  ~CSslProvider(void);

private:
  CSslCredentials m_sslCredentials;
  CNbtCriticalSection m_critSec;

public:
  // Intialization functions that must be
  // called prior to calling ObtainTransport()
  HRESULT Init(BOOL bAsServer, LPCTSTR strPrincipal,
    LPCTSTR strStore, BOOL bMutualAuth);
  HRESULT Init(BOOL bAsServer, LPCTSTR strPrincipal,
    HCERTSTORE hStore, BOOL bMutualAuth);
  HRESULT Init();
  void Destroy();
  HRESULT Attach(CredHandle hCredentials, BOOL bAsServer,
    BOOL bMutualAuth);
  HRESULT Detach();

  HRESULT ObtainTransport(CSslTransport*& pSslTransport);

  ... // other code not shown
};

... // other code not shown

// implementation

HRESULT CSslProvider::ObtainTransport(
          CSslTransport*& pSslTransport)
{
  m_critSec.Lock();

  HRESULT hr = S_OK;
  pSslTransport = NULL;

  // Make sure we obtained the credentials prior to this call
  if (m_sslCredentials.HasCredHandle())
  {
    pSslTransport = new CSslTransport(m_sslCredentials);
    if (pSslTransport == NULL)
      hr = E_OUTOFMEMORY;
  }
  else
    hr = SEC_E_NO_CREDENTIALS;

  m_critSec.Unlock();
  if (FAILED(hr))
    TRACE1("Failed to obtain transport. Error: 0x%08X.<\\>n", hr);

  return hr;
}
```

shortcuts to the function that acquires SChannel credentials. There, I'm simplifying the call to `AcquireCredentialsHandle()` with fewer parameters (see Listing 1 for details), which is sufficient enough for most of the cases. If the application you are developing requires a special treatment, you'll need to acquire the credentials yourself and then supply them in one of the `Init()` overrides.

The original `AcquireCredentialsHandle()` takes a lot of parameters (remember that this is a generic function, which is used by other types of security providers) but to initialize SChannel credentials only four of the parameters are relevant. Although "the fifth element," `ptsExpiry`, can optionally be used, SChannel always returns zero in it. I won't be wasting the reader's time describing each parameter except for one: the `SCHANNEL_CRED` structure. This is a rather important fellow through whom SChannel receives everything it needs to know about the principal, so let's scrutinize it.

SCHANNEL_CRED Structure and X.509 Certificates

When acquiring credentials for a server-side application, or for a client that is expected to present its credentials later in the SSL handshake, one very important member of the `SCHANNEL_CRED` structure should be looked at: `paCred`—a pointer to an array of `CERT_CONTEXT` structures (`cCreds`—the number of structures in the array). The `CERT_CONTEXT` structure contains an X.509 certificate [4] and the matching context in the form of a `HCERTSTORE`

Listing 2 CSslCredentials class

```
// other code not shown

// definition

class CSslCredentials
{
  friend class CSslProvider;

protected:
  CSslCredentials();
  ~CSslCredentials();

public:
  HRESULT Obtain(BOOL bAsServer, LPCTSTR strPrincipal,
    LPCTSTR strStore, BOOL bMutualAuth = FALSE,
    DWORD dwProtoFlags = 0);
  HRESULT Obtain(BOOL bAsServer, LPCTSTR strPrincipal,
    HCERTSTORE hStore, BOOL bMutualAuth = FALSE,
    DWORD dwProtoFlags = 0);
  HRESULT Obtain(DWORD dwProtoFlags = 0);

  HRESULT Attach(CredHandle hCredentials, BOOL bAsServer,
    BOOL bMutualAuth = FALSE, DWORD dwProtoFlags = 0);
  HRESULT Detach();

  BOOL IsServer() {return m_bServer;}
  BOOL MutualAuthRequired() {return m_bMutualAuth;}
  DWORD GetProtoFlags() {return m_dwProtoFlags;}

  BOOL HasCredHandle()
  {
    return SecIsValidHandle(&m_hCredentials);
  }

  PCredHandle GetHandle()
  {
    return (HasCredHandle() ? &m_hCredentials : NULL);
  }

  void CleanUp();

private:
  HRESULT OpenSysStore(LPCTSTR strStore, HCERTSTORE& hStore);
  HRESULT FindCertificate(HCERTSTORE hStore,
    LPCTSTR strPrincipal, PCCERT_CONTEXT& certContext);
  HRESULT ObtainImpl(PCCERT_CONTEXT certContext,
    BOOL bAsServer, BOOL bMutualAuth, DWORD dwProtoFlags);

private:
  CredHandle m_hCredentials;
  BOOL m_bServer;
  BOOL m_bMutualAuth;
  DWORD m_dwProtoFlags;
  HCERTSTORE m_hStore;

};

... // other code not shown

// implementation

HRESULT CSslCredentials::ObtainImpl(PCCERT_CONTEXT certContext,
                    BOOL bAsServer,
                    BOOL bMutualAuth,
                    DWORD dwProtoFlags)
{
  // Build Schannel credentials structure.
  SCHANNEL_CRED credSchannel = {0};
  credSchannel.dwVersion = SCHANNEL_CRED_VERSION;
  credSchannel.grbitEnabledProtocols = dwProtoFlags;

  if (certContext != NULL)
  {
    credSchannel.cCreds = 1;
    credSchannel.paCred = &certContext;
  }

  // Create SSL credentials.
  TimeStamp tsExpires;
  HRESULT hr = CSspiLib::AcquireCredentialsHandle(
      NULL,
      UNISP_NAME,
      (bAsServer ?
        SECPKG_CRED_INBOUND :
        SECPKG_CRED_OUTBOUND),
      NULL,
      &credSchannel,
      NULL,
      NULL,
      &m_hCredentials,
      &tsExpires);

  // See, if we have succeeded
  if (SUCCEEDED(hr))
  {
    m_bServer = bAsServer;
    if (m_bServer)
      m_bMutualAuth = bMutualAuth;
    m_dwProtoFlags = dwProtoFlags;
  }
  else
    TRACE(_T("Line: %d. Error: 0x%08X<\\>n"), __LINE__, hr);

  return hr;
}

... // other code not shown

HRESULT CSslCredentials::OpenSysStore(LPCTSTR strStore,
                      HCERTSTORE& hStore)
{
  CT2CA storeName(strStore);
  m_hStore = ::CertOpenSystemStore(0, storeName);
  if (m_hStore == NULL)
  {
    DWORD dwErrCode = ::GetLastError();
    return HRESULT_FROM_WIN32(dwErrCode);
  }

  return S_OK;
}

HRESULT CSslCredentials::FindCertificate(
          HCERTSTORE hStore,
          LPCTSTR strPrincipal,
          PCCERT_CONTEXT& certContext)
{
  USES_CONVERSION;
  CT2CA hostName(strPrincipal);
  ASSERT(m_hStore != NULL);
  certContext = ::CertFindCertificateInStore(
    m_hStore,
    X509_ASN_ENCODING | PKCS_7_ASN_ENCODING,
    0,
    CERT_FIND_SUBJECT_STR_A,
    hostName,
    NULL);

  if (certContext == NULL)
  {
    DWORD dwErrCode = ::GetLastError();
    return HRESULT_FROM_WIN32(dwErrCode);
  }

  return S_OK;
}
```

Listing 3 CsslTransport class

```
// other code not shown

// definition

[event_source(native)]
class CSslTransport
{
  friend class CSslProvider;

private:
  CSslTransport(CSslCredentials& sslCred);
  ~CSslTransport() {};

private: /*member variables*/

  ... // other code not shown
  CSslCredentials& m_sslCred;

  // SChannel parameters
  SecHandle m_hContext;
  SecPkgContext_StreamSizes m_streamSizes;

private: /*member functions*/

  // Actual data handling routines
  HRESULT ProcessPlainData();
  HRESULT ProcessCryptData();

  HRESULT ProcessExpiredContext(CNbtByteContainer& byteCont);
  HRESULT ProcessRenegotiate(CNbtByteContainer& byteCont);
  void ProcessDisconnect(CNbtByteContainer& byteCont);

  // SChannel helpers
  void ReleaseContext() {
    if (HasContext()) {
      CSspiLib::DeleteSecurityContext(&m_hContext);
      SecInvalidateHandle(&m_hContext);
    }
  }
  inline BOOL HasContext()
  {
    return SecIsValidHandle(&m_hContext);
  }
  HRESULT NegotiateAsSrvr(CNbtByteContainer& byteCont);
  HRESULT NegotiateAsClnt(CNbtByteContainer& byteCont);
  HRESULT DecryptData(CNbtByteContainer& byteCont);
  HRESULT EncryptDataOnce(PBYTE pDataIn, int iDataSizeIn,
    const SecPkgContext_StreamSizes &spcStreamSizes,
    CNbtByteContainer& byteCont);
  HRESULT EncryptData(CNbtByteContainer& byteCont);
  HRESULT SetSSLStreamSizes();

  inline void Lock() {m_critSec.Lock();}
  inline void Unlock() {m_critSec.Unlock();}

  // Events handling routines
  void FireOutboundDataAvail(CNbtByteContainer& byteCont);
  void FireInboundDataAvail(CNbtByteContainer& byteCont);
  void FireDisconnect(CNbtByteContainer& byteCont);

  // NOTE: native events (VC++7.0 only). If you're using VC6,
  // modify the code to use a callback routine
  __event void EventOutboundDataAvail(
    ULONG ulSize, BYTE* pbData);
  __event void EventInboundDataAvail(
    ULONG ulSize, BYTE* pbData);
  __event void EventDisconnect(
    ULONG ulSize, BYTE* pbData);

public: /*member functions*/

  // Data feeding routines
  HRESULT FeedPlainData(ULONG ulSize, BYTE* pbData);
  HRESULT FeedCryptData(ULONG ulSize, BYTE* pbData);
  HRESULT RequestDisconnect();

  void Destroy() {delete this;}

};

... // other code not shown

// implementation

HRESULT CSslTransport::NegotiateAsSrvr(
            CNbtByteContainer& byteCont)
{
  ASSERT(!m_bShuttingDown);
  ASSERT(m_sslCred.HasCredHandle());

    DWORD dwSspiInFlags =
          ASC_REQ_SEQUENCE_DETECT |
                    ASC_REQ_REPLAY_DETECT |
                    ASC_REQ_CONFIDENTIALITY |
                    ASC_REQ_EXTENDED_ERROR |
                    ASC_REQ_ALLOCATE_MEMORY |
                    ASC_REQ_STREAM;

  if (m_sslCred.MutualAuthRequired())
        dwSspiInFlags |= ASC_REQ_MUTUAL_AUTH;

    // Set Output Buffers for AcceptSecurityContext call
  SecBuffer dataOut[1];
  SecBufferDesc dataOutDesc;
  dataOutDesc.cBuffers = 1;
  dataOutDesc.pBuffers = dataOut;
  dataOutDesc.ulVersion = SECBUFFER_VERSION;

  SECURITY_STATUS scRet = SEC_I_CONTINUE_NEEDED;
  while (
    (scRet == SEC_I_CONTINUE_NEEDED) ||
    (scRet == SEC_E_INCOMPLETE_MESSAGE))
  {
    SecBuffer dataIn[2];
    dataIn[0].pvBuffer = m_dataCrypt.GetData();
    dataIn[0].cbBuffer =
      (unsigned long) m_dataCrypt.GetCount();
    dataIn[0].BufferType = SECBUFFER_TOKEN;

    dataIn[1].pvBuffer = NULL;
    dataIn[1].cbBuffer = 0;
    dataIn[1].BufferType = SECBUFFER_EMPTY;

    SecBufferDesc dataInDesc;
    dataInDesc.cBuffers = 2;
    dataInDesc.pBuffers = dataIn;
    dataInDesc.ulVersion = SECBUFFER_VERSION;

    dataOut[0].pvBuffer = NULL;
    dataOut[0].cbBuffer = 0;
    dataOut[0].BufferType = SECBUFFER_TOKEN;

    // Generate SSL handshake data
    TimeStamp tsExpire;
    DWORD dwSspiOutFlags = 0;
    scRet = CSspiLib::AcceptSecurityContext(
          m_sslCred.GetHandle(),
          (HasContext() ? &m_hContext : NULL),
          &dataInDesc,
          dwSspiInFlags,
          SECURITY_NATIVE_DREP,
          (HasContext() ? NULL : &m_hContext),
          &dataOutDesc,
          &dwSspiOutFlags,
          &tsExpire);

    ... // other code not shown

    if ((scRet == SEC_E_OK) ||
      (scRet == SEC_I_CONTINUE_NEEDED) ||
      ((FAILED(scRet) &&
      ((dwSspiOutFlags & ISC_RET_EXTENDED_ERROR) != 0))))
    {
      ... // other code not shown
      if ((dataOut[0].cbBuffer != 0) &&
        (dataOut[0].pvBuffer != NULL))
      {
        byteCont.Append((const BYTE*)
          dataOut[0].pvBuffer, dataOut[0].cbBuffer);
        CSspiLib::FreeContextBuffer(dataOut[0].pvBuffer);
        dataOut[0].pvBuffer = NULL;
        dataOut[0].cbBuffer = 0;

        // Done handshaking ?
        if (scRet == SEC_E_OK)
        {
          scRet = SetSSLStreamSizes();
          if (FAILED(scRet))
          {
            goto OnError;
          }

          if (dataIn[1].BufferType == SECBUFFER_EXTRA)
          {
            m_dataCrypt.RemoveAt(0, m_dataCrypt.GetCount() -
                                    dataIn[1].cbBuffer);
            return scRet;
          }
        }
        // Protocol Error?
        else if (FAILED(scRet) &&
          ((dwSspiOutFlags & ISC_RET_EXTENDED_ERROR) != 0))
        {
          return scRet;
        }
```

Listing 3 Continued

```
        m_dataCrypt.RemoveAll();
        return S_OK;
      }

      if ((scRet != SEC_E_INCOMPLETE_MESSAGE))
      {
        if (dataIn[1].BufferType == SECBUFFER_EXTRA)
        {
          m_dataCrypt.RemoveAt(0, m_dataCrypt.GetCount() -
                                  dataIn[1].cbBuffer);
          if (scRet == SEC_E_OK)
          {
            scRet = SetSSLStreamSizes();
            if (FAILED(scRet))
            {
              goto OnError;
            }

            return scRet;
          }
        }
        else if (scRet == SEC_I_CONTINUE_NEEDED)
        {
          m_dataCrypt.RemoveAll();
          return S_OK;
        }
      }

      ... // other code not shown

    }
    else
    {
      if (scRet == SEC_E_INCOMPLETE_MESSAGE)
        return SEC_E_INCOMPLETE_MESSAGE;
      goto OnError;
    }
  }

  scRet = SetSSLStreamSizes();
  if (FAILED(scRet))
  {
    goto OnError;
  }

  m_dataCrypt.RemoveAll();
  return S_OK;

OnError:
  m_dataCrypt.RemoveAll();
  return (HRESULT) scRet;
}

... // other code not shown

HRESULT CSslTransport::DecryptData(CNbtByteContainer& byteCont)
{
  ... // other code not shown

  const MaxBuffers = 4;
  SecBufferDesc dataInDesc;
  SecBuffer dataIn[MaxBuffers];
  memset(dataIn, 0, sizeof(dataIn));

  dataIn[0].pvBuffer = m_dataCrypt.GetData();
  dataIn[0].cbBuffer = (ULONG) m_dataCrypt.GetCount();
  dataIn[0].BufferType = SECBUFFER_DATA;

    dataInDesc.ulVersion = SECBUFFER_VERSION;
    dataInDesc.cBuffers = MaxBuffers;
    dataInDesc.pBuffers = dataIn;

  while (TRUE)
  {
    BOOL bExtra = FALSE;
    HRESULT scRet = CSspiLib::DecryptMessage(
                    &m_hContext, &dataInDesc, 0, NULL);
    switch (scRet)
    {
    case SEC_E_OK:
      try
      {
        for (int i = 0; i <\<> MaxBuffers; i++)
        {
          switch (dataIn[i].BufferType)
          {
          case SECBUFFER_DATA:
            byteCont.Append((const BYTE*) dataIn[i].pvBuffer,
                        dataIn[i].cbBuffer);
            break;

          case SECBUFFER_EXTRA:
            bExtra = TRUE;
            m_dataCrypt.RemoveAt(0, m_dataCrypt.GetCount() -
                        dataIn[i].cbBuffer);
            memset(dataIn, 0, sizeof(dataIn));
            dataIn[0].pvBuffer = m_dataCrypt.GetData();
            dataIn[0].cbBuffer = m_dataCrypt.GetCount();
            dataIn[0].BufferType = SECBUFFER_DATA;
            break;

          default:
            continue;
          }
        }
      }
      catch(...)
      {
        return E_OUTOFMEMORY;
      }

      if (!bExtra)
      {
        m_dataCrypt.RemoveAll();
        return S_OK;
      }

      break;

    case SEC_I_RENEGOTIATE:
      return ProcessRenegotiate(byteCont);

    case SEC_I_CONTEXT_EXPIRED:
      return ProcessExpiredContext(byteCont);

    default:
      return (HRESULT) scRet;
    }
  }

  return S_OK;
}

HRESULT CSslTransport::EncryptData(CNbtByteContainer& byteCont)
{
  ... // other code not shown

  HRESULT hr = S_OK;
  int iDataSize, iNoOfChunks, iLastChunkSize, i;
  if (m_streamSizes.cbMaximumMessage == 0)
  {
    return SEC_E_INTERNAL_ERROR;
  }

  byteCont.RemoveAll();
  iDataSize = m_dataPlain.GetCount();
  iNoOfChunks = iDataSize / m_streamSizes.cbMaximumMessage;
  iLastChunkSize = iDataSize % m_streamSizes.cbMaximumMessage;
  CNbtByteContainer contTemp;
  for (i = 0; i <\<> iNoOfChunks; i++)
  {
    hr = EncryptDataOnce(
            (PBYTE) (m_dataPlain.GetData() +
                    i * m_streamSizes.cbMaximumMessage),
            m_streamSizes.cbMaximumMessage,
            m_streamSizes,
            contTemp);
    if (FAILED(hr))
      break;

    byteCont.Append(contTemp);
  }

  if (SUCCEEDED(hr) && (iLastChunkSize <\>> 0))
    hr = EncryptDataOnce(
          (PBYTE) (m_dataPlain.GetData() +
                  i * m_streamSizes.cbMaximumMessage),
          iLastChunkSize,
          m_streamSizes,
          contTemp);

  m_dataPlain.RemoveAll();
  if (FAILED(hr))
  {
    byteCont.RemoveAll();
    if (hr == SEC_E_CONTEXT_EXPIRED)
    {
      hr = ProcessExpiredContext(byteCont);
    }
  }
  else
    byteCont.Append(contTemp);

  return hr;
}
```

handle. Typically, an application will have one set of the CERT_CONTEXT structures. If it's not enough, multiple certificate contexts can be loaded and specified.

A CERT_CONTEXT structure is obtained in the number of ways. In the accompanying sample application I'm extracting the certificate from the system's certificate store using the CSecCertificate wrapper class and its function—GetCertificate(). If you decide to go the same route, you'll need to open the store with CertOpenSystemStore() and look up the certificate by the principal's name with CertFindCertificateInStore(). By default Windows maintains four stores—MY, CA (Certification Authority), ROOT, and SPC (Software Publisher Certificates). You can examine them by going to the Internet Explorer properties dialog and clicking on the Content tab and then the Certificates button. Also by default, when you request a certificate from a Certification Authority (CA) that is based on Microsoft Certificate Server, Windows will add it to one of the system stores. Of course, there are ways to change that behavior. For example, you can create your own certificate store (look at certificate-related functions in [5]) and import the certificates. The store can be an internal file, or a registry blob, or what have you. If the software you are developing is for the in-house utilization only, then the best way to avoid higher costs associated with valid certificates issued by CA Root Authorities such as VeriSign is to install an in-house certificate server and establish your company's own Certification Authority. Certificate servers are beyond the scope of this article but you can easily install them on WinNT/2000/XP server platforms. For WinNT machines they can be downloaded and installed as part of the NT Option Pack, for 2000 and XP they are part of the system installation package. At the reader's disposal I provide a test certificate in the form of a PKCS#12 [6] file, which contains the certificate and the matching private key. To import the certificate and the key, simply double-click on the file and follow the wizard's directions. Be aware that the certificate has been issued and signed by Nebula Technologies CA, which is not part of the root certificate store on your machine. You'll have to suffer with an annoying dialog revealing that the certificate's path cannot be verified, although, this will suffice for testing purposes.

> Windows NT systems have security.dll exposing the SSPI functionality; all other Microsoft systems use secur32.dll

Finding the certificate in the system store will work if you already have a certificate and it was successfully imported into the store. Once the certificate is found your application may try to verify it before putting it to use. A more generic version, CertOpenStore(), can open a store that has been previously serialized into an external file. Or, it can open a PKCS#7 [7] file containing X.509 certificate chain. Another way is to manually create a certificate context from an existing X.509 certificate and add it to the previously opened store, thus obtaining both the certificate and the store context. The functions to look at are CertCreateCertificateContext() and CertAddCertificateContextToStore(). In all the above cases of acquiring the certificate, it's implied that the private key that corresponds to the certificate's public key exists in the security provider repository. Otherwise, your application will not be able to negotiate SSL read/write keys and calls to AcceptSecurityContext() or InitializeSecurityContext() will fail.

SSL Handshake and Data Exchange

Once AcquireCredentialsHandle() function successfully returns the handle, the next step in getting SChannel to do its work is to call AcceptSecurityContext() or InitializeSecurityContext() to perform the SSL handshake for us. The first function processes handshake messages as a server and the second one as a client. After that, EncryptMessage() and DecryptMessage() are used for just what their names impl —encrypting and decrypting messages. Feel free to look up these functions [5] and the meaning of their input/output parameters and return codes (which are plenty, by the way). In the supplemental code, CSslTransport class wraps these important functions. I sprinkled a lot of comments there in an attempt to explain what's behind all these SEC_E_INCOMPLETE_MESSAGE and SECBUFFER_EXTRA constants that inform the caller about the actions taken by SChannel or actions that the caller has to take on its part when the functions return.

Conclusion

I'm barely touching the tip of the iceberg of what's involved in creating a robust and responsive SSL-enabled client or server system, especially the server. If you anticipate a heavy load on that system, you should consider some optimization techniques. Eric Rescorla's book [8] outlines most of the problems related to SSL and TLS performance. The ability to support the maximum number of clients is of paramount importance for the server that anticipates heavy traffic. If you are using WinSock as the data supplier for the application, consider loading WinSock 2 and utilizing its bandwidth throttling capability [11].

To get a sense of how SChannel handles the data, try playing with the supplemental code available online. The supplemental code provides a good and verbose diagnostic feature. Good luck.

References

[1] Freier, A.O., Karlton, P., and Kocher, P.C., "The SSL Protocol Version 3.0." November 1996. This is the last published draft of SSLv3, which never materialized as an RFC but became the de facto standard. http://home.netscape.com/eng/ssl3/draft302.txt
[2] Dierks, T., Allen, C., "The TLS Protocol Version 1.0," RFC 2246. January 1999.
[3] ITU-T, "OSI networking and system aspects—Abstract Syntax Notation One (ASN.1)," ITU-T Recommendation X.690, December 1997.
[4] ITU-T, "Directory," ITU-T Recommendation X.509, August 1997.
[5] Microsoft, "MSDN Help." April 2002.
[6] RSA Laboratories, "Password Based Encryption Standard," PKCS #12. June 1999.
[7] RSA Laboratories, "Cryptographic Message Syntax Version 1.5," PKCS #7. November 1993.
[8] Rescorla, E., *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.
[9] Thomas, S., *SSL and TLS Essentials*. John Wiley & Sons, Inc., 1999.
[10] Linn, J., "Generic Security Service Application Program Interface (GSS-API)," RFC 1508. September 1993.
[11] Hua W., "Winsock 2: QoS API Fine-Tunes Networked App Throughput and Reliability." MSDN Magazine, Microsoft, April 2001.

w::d

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

Win32 Security in Managed C++

Defining the AccessToken class in MC++ and trying it out in a Security Explorer app

ALTHOUGH .NET PROVIDES ITS own security model (in the `System::Security` namespace) for CLR security, it doesn't address the lurking behemoth that is the Win32 security subsystem. For those who have programmed with Win32 security, you'll know that while it's conceptually straightforward, it's a real bear to program. For example, when writing a program that requires certain rights or privileges to work, it can often be a very intensive process to determine what privileges one has, or what rights are required.

In this article, I'll demonstrate how this coding headache can be simplified as I develop a .NET library, in Managed C++, that encapsulates the programming details in a simple object model. Along the way, we'll have a quick refresher on Win32 security, demonstrate some techniques for simplifying coding with the Win32 security APIs, and learn some of the restrictions and gotchas of Managed C++.

The article will be based on the first version of the .NET Win32 Security component I'm developing, and will describe the implementation of the `AccessToken` class and supporting types. It will also show how this object model can be utilized in a UI client, Security Explorer, written in C#.

MATTHEW WILSON *is a software development consultant for Synesis Software, specializing in robustness and performance in C, C++, C#, and Java. Matthew is the author of the STLSoft libraries, and the forthcoming book* Imperfect C++ *(to be published by Addison-Wesley, 2004). He can be contacted via matthew @synesis.com.au or at http://stlsoft.org/. This article was first published in* Windows Developer Network, *September 2003.*

Win32 Security Model Whistlestop

Win32 security is a big topic, which would require many articles to do it justice. (For those who want to learn more, I recommend Nik Okuntseff's book *Windows NT Security* as a good place to start.) I'm going to restrict myself in this article to a discussion of access tokens. An access token is created for each user when the user logs on, and persists for the duration of that user's session. Every process that the user executes receives a copy of that access token, which is then used by the security subsystem to determine what operations the process (i.e., the user) is allowed to perform.

An access token contains, among other things, the following items:

- The user's security identifier (SID). A SID is a variable-length binary structure that is used to uniquely identify entities (users and groups).
- A list of SIDs indicating the groups to which the user belongs.
- A default owner SID. This is assigned as the owning entity to objects that the user creates.
- A list of privileges. Note that some privileges must be enabled in order for them to be used.
- A default discretionary access control list (DACL). This is a list of permissions, known as access control entries (ACEs), that will be used when the user supplies a null `SECURITY_DESCRIPTOR` during system object creation (e.g., the fourth parameter in `CreateFile()`).

The creation and manipulation of some of these structures can be very cumbersome indeed. For example, an access control list (ACL) consists of a header—the Win32 structure ACL—followed by a variable number of access control entries (ACEs). Each ACE is a variable-length structure consisting of a header—`ACCESS_ALLOWED_ACE`, `ACCESS_DENIED_ACE`, or `SYSTEM_AUDIT_ACE`—all of which are (thankfully) the same size, and comprised of an `ACE_HEADER` structure, an `ACCESS_MASK`, and a SID (which is the variable length part). And that's just an ACL. A security descriptor, which is the security information applied to system objects (e.g., files, registry keys, processes, etc.) is comprised of two SIDs (owner and group) and two ACLs (permissions list and auditing list), all of which are variable length, and may or may not be comprised within the same memory block as the security descriptor or referenced via pointers. Scary, huh?

AccessToken

The `SynSoft::Security::AccessToken` class is defined, in Managed C++, as shown in Listing 1. The first thing you'll note is that it derives from the interface `System::IDisposable`. This interface marks the class as implementing the Dispose pattern, which means that the class can be used in a C# `using` expression (providing *Resource Acquisition Is Initialization*, albeit user-requested rather than class-mandated) and also is amenable to generalized closure: One simply dynamically casts (using `dynamic_cast` in C++ or `as` in C#) to `IDisposable`, then calls `Dispose()` on a (nonnull) resultant reference. `AccessToken`, like any class that owns/manages a native resource (in this case the access token) provides this facility so that well-behaved clients can cause it to relinquish its resources in a timely fashion, rather than waiting for garbage collection. Furthermore, it is my practice to provide a `Close()` method whenever implementing `IDisposable`, which performs the same task as `Dispose()`. You'll see this in other classes in this library.

Next, it is clear that one cannot construct an instance of an `AccessToken`. This makes sense, since we cannot create an access token in Win32 land—we can only open one via the Win32 functions `OpenProcessToken()` and `OpenThreadToken()`. `AccessToken` has two corresponding static methods that do exactly the same job, and then return an `AccessToken` instance representing the (successfully opened) token. So we can open a token and we can close/dispose of it. Now what we're interested in are its attributes.

The current version of this library contains read-only components. (Create/edit functionality is planned in a future release; visit http://synsoft.org/dotnet.html to obtain updates.) We can access, as .NET properties, the user and primary group, the groups, the privileges, the default DACL, and a variety of other attributes associated with an access token. The types returned (e.g., `SID`, `GroupList`, `PrivilegeList`) are also classes in the same namespace (and assembly). Most of these classes are also required to implement the `IDisposable` pattern; this is one of the hassles of .NET's not supporting C++'s deterministic destructors. Consider the `GroupList` class (Listing 2). This class does not contain any unmanaged resources. In fact, its only member is an ArrayList. However, because the list is populated with Group instances in the `GroupList` constructor, `GroupList` must implement the `IDisposable` interface, and in its `Dispose()` method must iterate through

Listing 1 The AccessToken class declaration

```
namespace SynSoft
{
  namespace Security
  {
    /// Represents a security AccessToken
    public __gc class AccessToken
      : public IDisposable
    {
    private:
      typedef void  *HANDLE;

    private:
      /// Create an instance managing the given security handle
      AccessToken(HANDLE hToken);
      /// Finaliser
      ~AccessToken();

    // IDisposable
    public:
      /// Provides facility for
      void  Close();
      void  Dispose();

    // Creation
    public:
      /// Open the access token for the current process
      static AccessToken *OpenProcessToken();
      /// Open the access token for the current thread
      static AccessToken *OpenThreadToken();

    // Attributes
    public:
      /// The token's user account
      __property SID *get_User();
      /// The groups accounts associated with the access token
      __property GroupList *get_Groups();
      /// Privileges
      __property PrivilegeList *get_Privileges();
      /// The security identifier of the owner
      __property SID *get_Owner();
      /// The primary groups of any objects created with the access token
      __property SID *get_PrimaryGroup();
      /// Default DACL
      __property ACL *get_DefaultDACL();
      /// Source
      __property TokenSource *get_Source();
      /// Type
      __property TokenType get_Type();
      /// Impersonation Level
      __property TokenImpersonationLevel get_ImpersonationLevel();
      /// Statistics
      __property TokenStatistics *get_Statistics();
      /// Restricted SIDs
      __property GroupList *get_RestrictedSIDs();
      /// Session Id
      __property Int32 get_SessionId();
      /// Sandbox Inert
      __property bool get_SandboxInert();

    // Standard members
    public:
      // Returns a string representation of the AccessToken
      String  *ToString();

    // Implementation
    private:
      GroupList *_get_groups(TOKEN_INFORMATION_CLASS tic);

    // Members
    private:
      HANDLE          m_hToken;
      SID             *m_user;
      GroupList       *m_groups;
      PrivilegeList   *m_privileges;
      SID             *m_owner;
      SID             *m_primaryGroup;
      ACL             *m_defaultDACL;
      TokenSource     *m_source;
      TokenStatistics *m_statistics;
      GroupList       *m_restrictedSIDs;
    };
  }
}
```

Listing 2 The GroupList class

```
namespace SynSoft
{
  namespace Security
  {
// GroupList.h

    /// Represents a list of groups
      public __gc class GroupList
      : public IEnumerable
      , public IDisposable
    {
    public:
      GroupList(int cGroups, SID_AND_ATTRIBUTES *groups);

    // IDisposable
    public:
      void  Close();
      void  Dispose();

    // IEnumerable
    public:
      IEnumerator *GetEnumerator();

    // Members
    private:
      ArrayList *m_groups;
    };

// GroupList.cpp

    GroupList::GroupList(int cGroups, SID_AND_ATTRIBUTES *groups)
      : m_groups(new ArrayList(cGroups))
    {
      for(int i = 0; i < cGroups; ++i)
      {
        m_groups->Add(new Group(new SID(groups[i].Sid), groups[i].Attributes));
      }
    }

    void GroupList::Close()
    {
      if(0 != m_groups)
      {
        dispose_contents(m_groups);
        m_groups->Clear();

        m_groups = 0;
      }
    }
  }
}
```

the list—ArrayList does not implement IDisposable—and call Dispose() on each contained object (in this case, Group instance). This stuff is tedious, and also carries a bit of a catch. Sometimes a class will implement IDisposable but will make its Dispose() method non-public; this is known as *Explicit Interface Member Implementation*. Hence, you need to cast the contained elements to IDisposable before attempting to call Dispose(). Because this is a common scenario, I've used the .netSTL (.netSTL is a subproject of STLSoft; http://dotnetstl.org/) algorithm dispose_contents<>() (shown in Listing 3, with its supporting function dispose_set_null<>()).

The picture doesn't end there, alas. Group also does not contain any unmanaged resources, but it has an instance of class SID, which does—a Win32 SID blob, in fact (see Listing 4). Thus, we had to add Dispose() (and Close()) functionality to at least two other types because a composed type requires it. Of course, you may think why bother, the Win32 SID is only a blob of memory (albeit from the native rather than the managed heap)? Well, as a rule, I don't feel comfortable having any unmanaged resources hanging around for an indefinite time, and one might ponder the case where the resource is not a block of memory but a file handle. This is a fundamental problem of garbage-collected systems; sure, they handle memory fine (often better than deterministically managed systems), but what about all the other resource types (files, pipes, and heaven-forefend, synchronization objects!)?

Having had my C++-biased rant, let's see what's good about this library. Looking into the implementation of many of the property accessor

Listing 3 IDisposable helper functions from .netSTL

```
/* ////////////////////////////////////////////////////////////
 * ...
 *
 * Extract from dotnetstl_dispose_functions.h
 *
 * Copyright (C) 2003, Synesis Software Pty Ltd.
 * (Licensed under the Synesis Software Standard Source License:
 *  http://www.synesis.com.au/licenses/ssssl.html)
 *
 * ...
 * //////////////////////////////////////////////////////////// */

namespace dotnetstl
{
  /// Disposes the managed type, and resets the pointer
  template <ds_typename_param_k T>
  inline void dispose_set_null(T *&pt)
  {
    if(0 != pt)
    {
      System::IDisposable *disposable = pt;

      disposable->Dispose();
      pt = 0;
    }
  }

  /// Disposes all the items in a container
  template <ds_typename_param_k C>
  inline void dispose_contents(C *pc)
  {
    for(int i = 0, count = pc->get_Count(); i < count; ++i)
    {
      System::IDisposable *o  = dynamic_cast<System::IDisposable*>(pc->get_Item(i));

      dispose_set_null(o);
    }
  }

} // namespace dotnetstl
```

Listing 4 Extracts from SID.h & SID.cpp

```
namespace SynSoft
{
  namespace Security
  {
// SID.h

    public __gc class SID
      : public IDisposable
    {
    public:
      SID(void *psid);
      ~SID();

    // IDisposable
    public:
      void  Close();
      void  Dispose();

    // Properties
    public:
      // Returns a string representation of the SID
      String *ToString();

      // The type of the SID
      __property SidType get_Type();
      // The name of the user associated with the SID
      __property String *get_UserName();
      // The name of the domain associated with the SID
      __property String *get_DomainName();

    // Members
    private:
      void    *m_psid;
      SidType m_type;
      String  *m_userName;
      String  *m_domainName;
      String  *m_stringForm;
    };

// SID.cpp

    SID::SID(void *psid)
      : m_psid(Sec_SidDup(psid))
      , m_type(SidType::Unknown)
      , m_userName(0)
      , m_domainName(0)
      , m_stringForm(0)
    {
      // Now we get the user and domain names

      if(0 == psid)
      {
        throw new SecurityException(E_INVALIDARG, "NULL SID");
      }
      else if(!::IsValidSid(psid))
      {
        throw new SecurityException(::GetLastError(), "Invalid SID");
      }
      else
      {
          . . .
      }
    }

    SID::~SID()
    {
      if(0 != m_psid)
      {
        Sec_Free(m_psid);
      }
    }

    void SID::Close()
    {
      if(0 != m_psid)
      {
        Sec_Free(m_psid);
        m_psid = 0;
      }
    }

    void SID::Dispose()
    {
      Close();
    }

    . . .
  }
}
```

methods of AccessToken (Listing 5), we can see one nice feature. Since all managed types must be held by pointer, the implementation dilemma common in object models—do I use *composition* to reduce memory usage and increase execution speed, or use *pimpl* to reduce memory usage and increase execution and compilation speed?—is moot. We have to use pointers, and it's often best to choose lazy evaluation. This may or may not be the "best" thing from a pure programming perspective, but it simplifies the design, coding, and maintenance of modules, so it has a lot going for it. A side effect is that it will help to ameliorate the dangling unmanaged resource issue; in cases where they are not used, they're not created, and so not dangling.

So how do we get information out of an access token? GetTokenInformation(), of course. This very powerful function is prototyped as follows:

```
BOOL WINAPI      GetTokenInformation (
  HANDLE                    TokenHandle,
  TOKEN_INFORMATION_CLASS   TokenInfClass,
  LPVOID                    TokenInformation,
  DWORD                     TokenInfLength,
  PDWORD                    ReturnLength);
```

You pass a token handle, the type of information you want (a member of the TOKEN_INFORMATION_CLASS enumeration), a destination buffer and its length (in bytes), and an address to receive the number of bytes of information retrieved. Seems straightforward, but like most Win32 Security API functions, it can be tricky. Some of the access token information is of a fixed size, and so one simply passes the appropriate parameters, as can be seen in the implementation of AccessToken::get_Type(). In most cases, however, the information is of variable length, so one must call GetTokenInformation() as described in the sidebar titled "Surviving the Win32 Security API." This approach is repetitive and error-prone—what if there is an exception thrown between the allocation and deallocation of the resource? A lot of careful boilerplate is required (see AccessToken::_get_groups() in Listing 5; this implementation is chosen because both TokenGroups and TokenRestrictedSids return information in a TOKEN_GROUPS structure). Mercifully, there is a simple and elegant solution in the form of the WinSTL (another STLSoft subproject; http://winstl.org/) component token_information<> (see Listing 6). You parameterize the template using the appropriate TOKEN_INFORMATION_CLASS enumeration member, and the traits mechanism automatically deduces the correct data structure type and

Listing 5 Extract from AccessToken.cpp

```
SID *AccessToken::get_User()
{
  if(0 == m_user)
  {
    token_information<TokenUser>  user(m_hToken);

    if(!user)
    {
      throw new SecurityException(::GetLastError());
    }
    else
    {
      m_user = new SID(user->User.Sid);
    }
  }

  return m_user;
}

GroupList *AccessToken::get_Groups()
{
  if(0 == m_groups)
  {
    m_groups = _get_groups(TokenGroups);
  }

  return m_groups;
}

GroupList *AccessToken::get_RestrictedSIDs()
{
  if(0 == m_restrictedSIDs)
  {
    m_restrictedSIDs = _get_groups(TokenRestrictedSids);
  }

  return m_restrictedSIDs;
}

PrivilegeList *AccessToken::get_Privileges()
{
  if(0 == m_privileges)
  {
    winstl::token_information<TokenPrivileges>  privileges(m_hToken);

    if(!privileges)
    {
      throw new SecurityException(::GetLastError());
    }
    else
    {
      m_privileges = new PrivilegeList(privileges->PrivilegeCount,
                          privileges->Privileges);
    }

    return m_privileges;
  }

  return m_privileges;
}

TokenType AccessToken::get_Type()
{
  TOKEN_TYPE  tt;
  DWORD       chRequired;

  if(!::GetTokenInformation(m_hToken, ::TokenType,
                      &tt, sizeof(tt), &chRequired))
  {
    throw new SecurityException(::GetLastError(),
                      "Could not elicit type from token");
  }

  return TokenType(tt);
}

GroupList *AccessToken::_get_groups(TOKEN_INFORMATION_CLASS tic)
{
  GroupList *grouplist;
  DWORD     cbRequired;
  DWORD     dwErr;

  stlsoft_assert(tic == TokenGroups || tic == TokenRestrictedSids);

  ::GetTokenInformation(m_hToken, tic, NULL, 0, &cbRequired);
  dwErr = ::GetLastError();

  if(ERROR_INSUFFICIENT_BUFFER != dwErr)
  {
    throw new SecurityException(dwErr);
  }
  else
  {
    TOKEN_GROUPS  *groups = static_cast<TOKEN_GROUPS*>(Sec_Alloc(cbRequired));

    if(!::GetTokenInformation(m_hToken, tic, groups, cbRequired, &cbRequired))
    {
      throw new SecurityException(::GetLastError());
    }
    else
    {
      try
      {
        grouplist = new GroupList(groups->GroupCount, groups->Groups);
      }
      catch(Exception *x)
      {
        Sec_Free(groups);

        throw x;
      }
    }
  }

  return grouplist;
}
```

instantiates a class that wraps the allocation, acting as a smart pointer for the requisite type. The `get_User()` and `get_Privileges()` methods in Listing 5 demonstrate how much this simplifies matters. (Note that the template is flexible in allowing you to specify an exception policy. The default is `stlsoft::null_exception`, which does not throw, in which case the implementation ensures that `GetLastError()` will reflect the failing condition, rather than being overwritten during deallocation of the memory buffer.)

The difficulty of processing the information returned depends on the type of information retrieved. Retrieving `SECURITY_IMPERSONATION_LEVEL`, which is an enumeration (as is `TOKEN_TYPE`), is very simple. Just retrieve it and convert it to the appropriate managed enumeration type. Note: It's convenient, but somewhat unsafe, that Managed C++ lets you cast, in the form of a conversion constructor, from a C-type to a managed enumeration type; see `AccessToken::get_Type()`. Although I've borrowed the values for the managed enumerations `TokenType`, `TokenImpersonationLevel`, and `SidType` from their native equivalents, and am confident I've transcribed them correctly, what if I got it wrong?)

For some variable sized entities, it is also reasonably easy to process the retrieved information. Retrieving groups or privileges returns

Surviving the Win32 Security API

IF THERE are two things you should remember above all others when working with the Win32 Security API, they are:

1. Nearly everything is variable length. Not only does this mean you should make no assumptions about being able to iterate through composite/collection types in an integral fashion, but you should also never assume that things are of a predeterminable size.

2. The common paradigm is *Size-Allocate-Retrieve*, as we saw with `GetTokenInformation()`. You call a function with a null destination pointer and give an initial size of 0. It will return into your size parameter the required size (and set the last error to `ERROR_INSUFFICIENT_BUFFER`). You allocate an appropriately sized buffer and call again, and the function will fill your buffer with the desired information.

—M.W.

Listing 6 WinSTL's token_information helper template

```
/* ////////////////////////////////////////////////////////////////////
 * ...
 *
 * Extract from winstl_token_information.h
 *
 * Copyright (C) 2003, Synesis Software Pty Ltd.
 * (Licensed under the Synesis Software Standard Source License:
 *  http://www.synesis.com.au/licenses/ssssl.html)
 *
 * ...
 * ////////////////////////////////////////////////////////////////// */

namespace winstl
{
  template <TOKEN_INFORMATION_CLASS C>
  struct token_information_traits;


  template <>
  struct token_information_traits<TokenUser>
  {
    typedef TOKEN_USER  data_type;
  };

  ... // More specialisations of token_information_traits

  template< TOKEN_INFORMATION_CLASS C
          , ws_typename_param_k      X = stlsoft::null_exception
          , ws_typename_param_k      D = token_information_traits<C>::data_type
          , ws_typename_param_k      A = processheap_allocator<ss_byte_t>
          >
  class token_information
  {
  public:
    typedef token_information<C, X, D, A> class_type;
    typedef token_information_traits<C>   traits_type;
    typedef X                             exception_thrower_type;
    typedef D                             data_type;
    typedef A                             allocator_type;

  // Construction
  public:
    /// Constructs an instance from the given access token
    ws_explicit_k token_information(HANDLE hToken)
      : m_data(0)
    {
      DWORD cbRequired;
      DWORD dwError;

      ::GetTokenInformation(hToken, C, NULL, 0, &cbRequired);
      dwError = ::GetLastError();
      if(ERROR_INSUFFICIENT_BUFFER != dwError)
      {
        // Report error
        exception_thrower_type()(dwError);
      }
      else
      {
        data_type *data = (data_type*)allocator_type().allocate(cbRequired);

        if(NULL == data)
        {
          // Report error
          exception_thrower_type()(ERROR_NOT_ENOUGH_MEMORY);

          // Set the last error, in case the client code is not
          // using exception reporting
          ::SetLastError(ERROR_NOT_ENOUGH_MEMORY);
        }
        else
        {
          if(!::GetTokenInformation(hToken, C, data, cbRequired, &cbRequired))
          {
            // Scope the last error, in case the client code is not
            // using exception reporting
            dwError = ::GetLastError();

            allocator_type().deallocate((ss_byte_t*)data);

            // Report error
            ::SetLastError(dwError);
            exception_thrower_type()(dwError);
          }
          else
          {
            // Success
            m_data = data;

            ::SetLastError(ERROR_SUCCESS);
          }
        }
      }
    }
    ~token_information()
    {
      allocator_type().deallocate((ss_byte_t*)m_data);
    }

  // Conversion
  public:
    operator data_type *();
    operator data_type const *() const;

    data_type *operator ->();
    data_type const *operator ->() const;

    ws_bool_t operator !() const;

  /// Members
  private:
    data_type *m_data;

  /// Not to be implemented
  private:
    token_information(token_information const &);
    token_information &operator =(token_information const &);
  };

} // namespace winstl
```

counted arrays of SID_AND_ATTRIBUTES and LUID_AND_ATTRIBUTES, respectively, so one need simply walk the array, as in the GroupList constructor (Listing 2).

Access Control Lists and Entries

The retrieval of the default DACL from an access token is an altogether more complex matter. As well as retrieving a variable length structure, the items within the structure are polymorphic (can be one of the three ACE types), and are themselves of variable length. Nasty.

The answer comes from a combination of another useful class from WinSTL—the acl_sequence<> template (not shown; available from http://winstl.org/downloads.html)—and a hacky assumption. Since all the ACE structures have the same essential structure (which can't be a coincidence—I bet the designers of the Win32 Security API decided in this one small way to give themselves a break), we can treat them as sharing a pseudotype (wherein ACE_HEADER) to treat them all equally. Listing 7 shows the solution in the ACL constructor. acl_sequence<> takes care of defining the range [begin, end), and ensuring the appropriate pointer-arithmetic when moving from one ACE to the next. Inside the loop, we switch on the header type, and then call the appropriate ACE constructor. AccessAllowedACE, AccessDeniedACE, and SystemAuditACE share a common base class, ACE, which handles the retrieval of the ACE structure contents into the class in a generalized (based on their common layouts) fashion. As I said, it's a hack, but since it would be impossible to change the sizes of the extant ACE types without breaking every Win32 program out there that uses security, I think we can say it is an informed hack and sleep soundly.

Security Explorer

That's pretty much all the ugly stuff in the library; the rest is very straightforward and I'll leave it to you to download the archive and look through it. Let's see how we can use it. The Security Explorer program is shown

Strings in Managed C++

CONVERTING FROM a C-string to a managed string is nice and straightforward, as System::String contains appropriate constructors. However, going the other way is a bit of a pain (and it is not particularly well publicized in the documentation). You need to call the StringToHGlobalUni() or StringToHGlobalAnsi() methods of System::Runtime::InteropServices::Marshal type to allocate it, followed by ToPointer() on the IntPtr instance returned by them. When you're done with the string, don't forget to release it by calling Marshal's FreeHGlobal() method:

```
wchar_t *pwszUserName =
(wchar_t*)(Marshal::StringToHGlobalUni(userName).ToPointer());
size_t  si;
PSID    psid = Sec_AllocSidFromUserNameW(pwszUserName, &si);

Marshal::FreeHGlobal(pwszUserName);
```

Another thing worth mentioning is that just because you're in Managed C++, there's no reason why you must eschew all of your useful C/C++ libraries. Specifically, it's surprising how C++ aficionados forget the incredible utility of sprintf() and start flailing around with ToString() and Concat(). Consider the following possible implementations of LUID::ToString() (an LUID—Locally Unique ID—is used to uniquely identify a privilege, like an ATOM does for strings):

```
String *LUID::ToString()
{
  return String::Concat(HighPart.ToString(),
                "-",
                 LowPart.ToString());
}

String *LUID::ToString()
{
  wchar_t sz[31];

  wsprintfW(sz, L"%d-%u", HighPart, LowPart);

  return new String(sz);
}
```

The first one might have fewer lines, but which one do you think will be faster (and have fewer allocations)?

—M.W.

Listing 7 Enumerating an ACL

```
// from ACL.h

namespace SynSoft
{
  namespace Security
  {
    public __gc class ACL
      : public IEnumerable
      , public IDisposable
    {
    public:
      ACL(PACL pacl);

    // IDisposable
    public:
      void  Close();
      void  Dispose();

    // IEnumerable
    public:
      IEnumerator* GetEnumerator();

    // Properties
    public:
      __property Int32 get_Count();

      String  *ToString();

    // Members
    private:
      UInt32    m_revision;
      ArrayList *m_aces;
    };
  }
}

// GroupList.cpp

ACL::ACL(PACL pacl)
  : m_aces(new ArrayList(pacl->AceCount))
  , m_revision(pacl->AclRevision)
{
using winstl::acl_sequence;

acl_sequence            aces(pacl);
acl_sequence::const_iterator  begin = aces.begin();
acl_sequence::const_iterator  end   = aces.end();

  for(; begin != end; ++begin)
  {
    PACE_HEADER header  = *begin;

    switch(header->AceType)
    {
      case  ACCESS_ALLOWED_ACE_TYPE:
        m_aces->Add(new AccessAllowedACE(header));
        break;
      case  ACCESS_DENIED_ACE_TYPE:
        m_aces->Add(new AccessDeniedACE(header));
        break;
      case  SYSTEM_AUDIT_ACE_TYPE:
        m_aces->Add(new SystemAuditACE(header));
        break;
    }
  }
}
```

in Figure 1. As you can see, it has a standard tree/list format, where the tree contains composite or collection entities, and the list contains the values or attributes. This first version allows exploration of the process access token. The root-most node, when selected, displays the user, owner, primary group, and the like, and then the five subnodes show the contents of each of their collections.

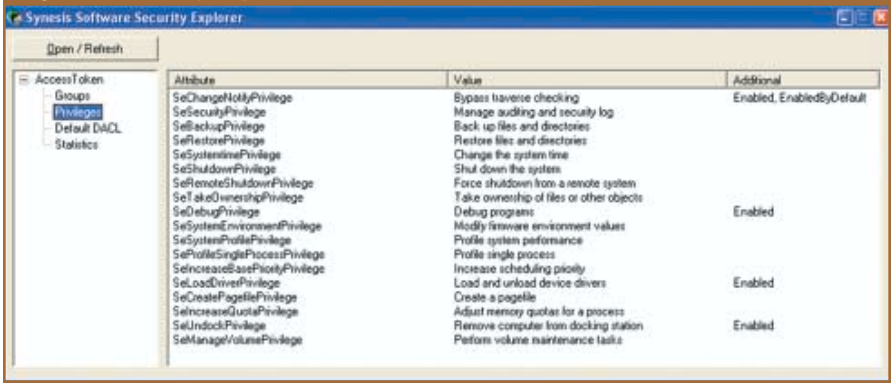
The Security Explorer works by opening the access token for the process, by calling `AccessToken::OpenProcessToken`. It stores this along with an identifying type in an instance of a `NodeItem` type, which is then stored as the `Tag` value (`LPARAM` in Win32 API parlance) of the tree node. The collection properties (`Groups`, `Privileges`, etc.) are similarly stored in subnodes. When the user clicks on a node, the handler loads the stored object (and its type), and processes it according to the type. It's not particularly elegant—a more professional solution would be to use polymorphic display-handler classes—but it serves to exercise the security object model. I'm already finding this useful, since it's far and away the quickest mechanism I have to tell me what privileges I have, and whether they're enabled, on a given system.

Summary

This article has given a very brief introduction to the Win32 security subsystem, discussing in some detail the contents of access tokens and how to read them. Hopefully, this has whetted your appetite and shown you that, in combination with suitable helper code, manipulating the security API is not as scary as you might have thought. We all avoid it like the plague as a rule, but when you need to secure your application objects, you have no choice but to dive in. Now, perhaps, you'll have a bit of background info to help.

We've also discussed some of the implications of development in Managed C++, and highlighted some of the issues you encounter with this new language. Finally, I hope the point is not lost that we've tamed a daunting C API by applying some very modern C++ (STL) techniques. Over this, we've layered a Managed C++ object model, which was then used in a C# application. How's that for integration?

Figure 1 Security Explorer Interface



Notes & References

Applied .NET Framework Programming. Jeffrey Richter, Microsoft Press, 2002.

Windows NT Security. Nik Okuntseff, R & D Books, 1997.

STLSoft is an open-source organization applying STL techniques to a variety of technologies and operating systems. It is located at http://stlsoft.org/. WinSTL and .netSTL are subprojects, and are located at http://winstl.org/ and http://dotnetstl.org/, respectively.

SynSoft is the nonprofit imprint of my company, Synesis Software, and is located at http://synsoft.org/. It provides D, .NET, Java, Perl, and Python code/components free for use without royalty.**w::d**

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

Managed C++ Hasn't Got Any Friends!

WHILE MANAGED C++ is, for the most part, easy to use (probably easier than pure C++ itself), it buys this by making some sacrifices. There are, therefore, some really annoying facets of the language:

Managed C++ has no friends. This might seem a minor thing, especially since use of the `friend` keyword is so rarely useful/justified in pure C++. However, there are other restrictions of managed C++ that conspire with the lack of friendship to create a real wart. Take the `SID` class. Its constructor (which takes `void*`) is visible, since we have to be able to make `SID` instances from within the `ACE`, `AccessToken`, and `GroupList` classes. We most certainly do not want any code outside of this library to ever call that constructor. However, if I make the constructor private, the other classes cannot see it. If I try to make them friends, I'm given a C3809 and told to go away. I even tried placing the address of a private static creator method in a public function pointer within a static (class) constructor, but this was disallowed because it violated accessibility to managed types from unmanaged types. Unless Managed C++ supports the Assembly or Family and Assembly accessibilities—which it does not, as far as I can discover—then we're stuck with situations such as that with `SID(void*)`. What's a J# client going to make of that?!

I mentioned in the main text that we can conversion-construct managed enumeration instances from native types, which is very open to abuse. Given that, it is strange that, even in managed C++, we cannot define constructors for value types. The `LUID` type, which is defined as:

```
public __value struct LUID
{
public:
  String *ToString();

public:
  void Assign(::LUID const &rhs)
  {
    this->LowPart  = rhs.LowPart;
    this->HighPart = rhs.HighPart;
  }

public:
  UInt32  LowPart;
  Int32   HighPart;
};
```

has to use the Assign method because we cannot define a constructor to instantiate it from the Win32 LUID structure. I presume there's a good reason, but it's annoying nonetheless.

—M.W.

Analyzing the Mescaline Worm

The Mescaline worm can be stopped with the right system patches, but stopping similar worms in the future requires a deeper look at the default settings that make these attacks possible

A WORM IS ABOUT to destroy the Internet. Again. For the second time this year. The worm has been given a name in advance this time, like tropical storms that are sure to develop each season before a few become hurricanes. The name of this worm shall be Mescaline (http://securityfocus.com/columnists/1741). It will target the now well-known vulnerabilities in Distributed Component Object Model (DCOM) that can be invoked remotely by way of Remote Procedure Call (RPC), as described in Microsoft Security Bulletin MS03-026 (http://www.microsoft.com/technet/security/bulletin/MS03-026.asp) and demonstrated by a variety of sample code and scanning utilities; see Resources.

Beating Mescaline before it strikes is now a matter of patching Windows boxes and protecting vulnerable TCP/IP ports. The vulnerable DCOM code can be reached in many ways, both locally and remotely, such as through ports 135 (RPC), 139 (NetBIOS), and 445 (SMB) that are exposed by default on Windows boxes. Other TCP/IP ports also expose an interface to DCOM in configurations where services like RPC over HTTP (port 593) are available or in the case of nearly any custom code that uses DCOM for interprocess communication. Just as we would prepare our homes and businesses to weather a coming storm, first by building a secure foundation, sturdy frame, and reliable roof, and next by covering breakable glass and other unpreventable vulnerabilities with plywood until the storm passes, we all must stop whatever else we're doing and hurry to patch, protect, and repair our Windows boxes. While doing so, we should figure out how to prevent our work from causing this type of problem again so future storms remain small and the Internet weather bureau won't have to make a regular practice of naming things in advance.

Note that the line of code that enables Mescaline isn't in DCOM, rather, it's in the network communications portion of RPC. The defective line of code causes RPC to bind TCP port 135 on the address of the default interface (0.0.0.0), which is also known as "all unassigned" or "INADDR_ANY." This binding is done instead of explicit bindings to each address supported by the box. This conserves memory, improves performance, and ensures that hardware configuration changes such as adding or removing network adapters that result in readdressing, or in the case of DHCP a preplanned dynamic addressing, don't result in network services that lose track of the current address to which they must bind in order to receive data over the network. A TCP/IP stack is also memory and performance sensitive, and conventional wisdom says that you err on the side of the small footprint when building a general-purpose network service like RPC that might need to communicate on many, if not all, network interfaces present on a box. As is often the case, this conventional wisdom is just plain wrong when you consider its security impact.

A Windows box should not expose features that may allow remote execution of code, either intentionally or because of bugs, unless the Windows box is actively controlled, monitored, protected, and patched by its rightful owner. Mescaline would be virtually impossible today, despite the DCOM buffer overflow bug, if Microsoft had chosen not to force Windows to expose ports 135, 139, and 445 as listening TCP/IP endpoints. Microsoft made it extremely difficult, and in some cases impossible, to disable system components like RPC that bind these ports. Our software should only expose features we actually use when it binds to network endpoints.

In order to use typical enterprise software packages under Windows, you must leave the default Windows services active, including components that expose remote services, even though you may never use these features remotely. They are often called locally as a mechanism for interprocess communication (IPC). The fact that RPC can be used instead of the Local Procedure Call (LPC) interface for IPC results in programmers building software that uses RPC when it should use LPC. They get away with it because it works.

No registry settings or other configuration options can force RPC to stop binding to the default interface. This is apparently by design. Unsupported registry hacks are possible that remove all network interfaces while preserving some aspects of network functionality, or that simply bind the majority of services to the loopback adapter where they can only be addressed locally (IIS Security and Programming Countermeasures, Ch. 4: Platform Security, http://www.forensics.org/jasonc/iisforensics.zip). A supported registry setting was introduced in a Windows 2000 service pack, however, that will compel Windows to stop binding port 445 entirely. The registry entry is a DWORD of 0 (disable) or 1 (enable) located at the following key/value:

```
HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001
    \Services\NetBT\Parameters
    \ SMBDeviceEnabled
```

Disabling DCOM using dcomcnfg.exe helps to protect against remote exploits of the MS03-026 DCOM RPC vulnerability, but considering the extent to which Windows aggressively compels the presence of this type of system service, there are most likely other ways for it to be called, at least locally. MS03-026 is precisely the type of bug that is prone to return if service packs or hotfixes are reapplied in the future. As programmers, we must take a moment to appreciate the technical and business process failures that led to this vulnerability. A postmortem review of other people's compiled code isn't nearly as helpful to our collective knowledge of information security as is a detailed review of the vulnerable source code. Vendors of compiled code whose software is found to contain security flaws should publish a thorough forensic analysis of the source code that produced the vulnerability. Full disclosure both educates other programmers and reassures customers that the full extent of the vulnerability has been assessed accurately. Without full disclosure, and without access to the source code, we have no reason to trust the patched code any more than we trusted the original. And we can't disable its features.

Resources

http://www.securiteam.com/exploits/5CP0N0KAKK.html

http://www.iss.net/support/product_utilities/ms03-026rpc.php

http://www.eeye.com/html/Research/Tools/RPCDCOM.html

http://www.lsd-pl.net/files/get?WINDOWS/win32_dcom w::d

JASON COOMBS *works as a forensic analyst and expert witness in court cases involving digital evidence. Information security and network programming are his areas of special expertise. He can be reached at jasonc@science.org.*

JASON COOMBS

Backdoors Can Damage Trust

Some backdoor security holes are unintentional bugs, but any backdoor should be analyzed with the view that the defect may have been purposeful

INFORMATION SECURITY VULNERABILITIES that result in backdoors to trusted systems are serious threats even when those systems don't store sensitive data. A trusted system holds the potential for harmful transgressions against both local users, who trust that their commands will be executed in a consistent, predictable manner, and remote users of other systems who presume that a trusted system is well behaved because it is under the control of trustworthy people and software.

A backdoor introduces far-reaching uncertainty both in terms of actual damage that may be caused when exploited, and in terms of intangible damage to trust. There is simply no way to know how many times the backdoor was exploited in the past, or to know precisely who created the backdoor and who knew about it and when.

Some backdoors are easy to find by reading source code because they are intentional. A branch of logic that explicitly bypasses security checks or improperly grants elevated privileges isn't hard to spot. These obvious backdoors can be easily prevented with good quality-assurance methods including a source-code review and proper system design documentation. We must also be willing to presume that our coworkers or employees are not trustworthy and may intentionally plant backdoors, which can be an uncomfortable test of mutual respect and trust itself.

The real problem, technically, is that unintentional backdoors caused by exploitable bugs, like unchecked buffer lengths inside input parsing routines, can result in buffer overflow conditions that are indistinguishable from intentional backdoors. It is safe to assume that most backdoors are written unwittingly, by programmers who aren't trying to get fired, prosecuted, or lynched by an angry mob.

A subtle bug in just the right place will result in a secret backdoor that can survive in a codebase for years even as QA and security audit duties pass from person to person or company to company. Explicit, obvious backdoors coded as extraneous branches of logic not anticipated during systems analysis shouldn't survive long enough to pass QA testing and end up deployed to production boxes, but they can and do in the real world. Backdoors that are obvious when source code is analyzed are particularly easy for programmers to add to software when the programmers themselves are also the QA engineers and security auditors.

Though you may never have conducted security audits of source code written by other people, you can imagine how easy it would be to overlook a single bug written on purpose and concealed using some clever coding technique. You may also be able to code and conceal such bugs yourself, in just a few minutes, that you would probably overlook if not for the fact that you designed them.

Perfect source code is only a contributing factor to secure software, it doesn't directly result in a security-hardened, trustworthy software product. After ensuring that source code is security-hardened to perfection, you must complete the highly vulnerable process of compiling, linking, copying, and distributing executable binaries each time you wish to turn source code into software. Bugs or intentional malfeasance at any step in this process could add a backdoor to the compiled code with a change to just a single bit.

When the following example code is compiled properly and then executed, it displays two 12-byte ASCII greetings: "Hello World!" and "Hello Buffer". This code results in software that is free of backdoors only if the compiler and linker work as expected and the binary executable file is not tampered with prior to its use. The length of input bytes is carefully limited by the code to prevent a buffer overflow from occurring, unless a vulnerability is intentionally introduced using a binary image editor (see HexEdit from http://www.expertcomsoft.com).

```
#include "stdafx.h"
#include <string.h>
#include <windows.h>

void func(char *inbuf, int buflen);

int main(int argc, char* argv[]) {
 char * hellow = "Hello World!";
 func(hellow,strlen(hellow));
 char * hellobof = "Hello Buffer Overflow!";
 func(hellobof,strlen(hellobof));
 return 0;
}

void func(char *inbuf, int inbuflen) {
 char malbuf[13]; // sized + 1 for null
 if(inbuflen >= 13) {
  inbuflen = 12; // max input buffer array
       index
 }
 malbuf[inbuflen] = NULL; // string terminator
 CopyMemory(malbuf,inbuf,inbuflen);
 printf(malbuf);
 printf("\n");
 return;
}
```

The hard-coded buffer array index of 12 (0C) must be set correctly during the function call at run time whenever the input length bounds check branch is traversed. Otherwise, a stack buffer overflow may occur when too many bytes are supplied by the caller. If you look at the machine code instructions produced by the compiler for this function call, you can see that a change to a single higher-order bit in the value stored to `inbuflen` would allow excessive input data to be stored beyond the bounds of the input buffer.

The compiler preallocates memory of a fixed length within the stack frame that is set up at run time for each call to the function. The programmer, aware of the potential for a stack buffer overflow in this circumstance if the source code does not include an explicit check on the input data length compared to the input buffer size, restricts the buffer copy to the potentially smaller fixed length hard-coded in the source. Software built from this source code should now be "safe" in terms of being free from backdoors. Until a single bit is changed

in the resulting machine code that introduces a mismatch between the size of the stack buffer used to receive the copy of the input buffer bytes and the length-checking bounds protection added by the programmer to prevent buffer overflows.

A single bit modification to the machine code results in a buffer overflow vulnerability that is likely to be exploitable as a backdoor if code like this appears in a program that exposes any sort of interprocess request processing feature. In the machine code tampering shown below, the hard-coded maximum array index value of 12 (0C) is replaced with 28 (1C) and the program, when executed, displays the full "Hello Buffer Overflow!" greeting before it crashes due to the damage caused to the current stack frame by the overflow.

Original machine code in hex:

```
00401040: 8B 4C 24 08 83 EC 10 83 F9 0D 7C 05
    B9 0C 00 00
00401050: 00 C6 44 0C 00 00 56 8B 74 24 18 8B
    C1 57 8D 7C
```

Modified machine code in hex:

```
00401040: 8B 4C 24 08 83 EC 10 83 F9 0D 7C 05
    B9 1C 00 00
00401050: 00 C6 44 0C 00 00 56 8B 74 24 18 8B
    C1 57 8D 7C
```

Vulnerabilities are like hidden features just waiting to be called upon by a knowledgeable power user. Microsoft's .NET Framework, Java's Virtual Machine, and other type-safe bytecode-based run-time environments are designed to minimize the risk of hidden features in software. By producing Microsoft Intermediate Language (MSIL) instead of native machine code, Microsoft .NET compilers attempt to create byte code that is usable at run time just like software, but that can't compromise or circumvent the security policy of its host—even if the byte code is tampered with by an attacker.

The following C# programs show a network service that attempts to expose a buffer overflow style backdoor on purpose, and an exploit program that attempts to overflow the receive buffer as would be necessary to take advantage of the hidden backdoor feature. Attempt to modify the MSIL byte code produced by the C# compiler in order to introduce the same type of buffer overflow vulnerability as illustrated previously. You'll find that the most you can do, without discovering an exploitable bug/backdoor in the .NET runtime itself, is cause valid changes to the program's MSIL code or break the program completely. There is no immediate potential to add an exploitable backdoor with a change to a single bit as there is when working directly with native machine-code compilations.

```
using System;
using System.Net.Sockets;
using System.Text;

namespace csharpbackdoor {
 class backdoorclass {
  [STAThread]
  static void Main(string[] args) {
   byte[] malicious;
   int bytes;
   bool loop = true;
   TcpClient tcp;
   NetworkStream net;
   TcpListener listen;
   listen = new TcpListener(4444);
   listen.Start();
   while(loop) { try {
    tcp = listen.AcceptTcpClient();
    net = tcp.GetStream();
    malicious = new byte[133];
    bytes = net.Read(malicious, 0, 133);
    System.Console.WriteLine(Encoding.ASCII.
        GetString(malicious));
    tcp.Close(); }
    catch (Exception readerror) {

System.Console.WriteLine(readerror.Message);
    loop = false; }}}}
}
```

Note hex 85 occurs in two places in the MSIL that results from compiling `csharpbackdoor`—this corresponds to decimal 133, and the second occurrence would be your target with your binary editor if you wanted to try to force the `Read()` function to overflow the fixed-length 133-byte buffer `malicious`. You can use `dumpbin.exe /all` to take a look at the MSIL yourself and observe the following:

```
RAW DATA #1
00402080: 0A 13 04 20 85 00 00 00 8D 13 00 00
01 0A 11 04
00402090: 06 16 20 85 00 00 00 6F 13 00 00 0A
0B 28 14 00
```

Modify the MSIL all you want. Any change to the second value (85 00 00 00) that causes it to exceed the first value (85 00 00 00) will result in a runtime exception rather than an overflow. Change both values to the same larger or smaller number and you will grow or shrink the receive buffer but otherwise the program will still operate. Use the `exploitbackdoor` program to attempt to stuff more bytes into the receive buffer than it is sized to hold. Tweak the MSIL for the backdoor program using HexEdit and then try the exploit again. Repeat until C# and .NET give you that warm fuzzy feeling of security complacency.

```
using System;
using System.Net.Sockets;
using System.Text;

namespace exploitbackdoor {
class backdoorexploit {
[STAThread]
static void Main(string[] args) {
 String s = String.Empty;
 if(args.Length > 0) {
  s = s.PadRight(int.Parse(args[0]),'A');
 }
 else {
  s = s.PadRight(2048,'A');
 }
 byte[] overflow = new byte[s.Length];
 int bytes = 0;
 System.Console.WriteLine("Sending: " + s);
 bytes = Encoding.ASCII.GetBytes
        (s,0,s.Length,overflow,0);
 TcpClient tcp = new TcpClient();
 try {
  tcp.Connect("localhost", 4444);
  NetworkStream networkStream =
tcp.GetStream();
  networkStream.Write(overflow, 0,
overflow.Length);
  tcp.Close(); }
 catch (Exception writeerror) {
  System.Console.WriteLine(writeerror.Message);
}}}}
```

The .NET Framework verifies at run time that the receive buffer is large enough to hold the data that the call to the `Read()` function will attempt to place there. Because the program is compiled to MSIL, a change to the byte code is not supposed to cause any security problem. We can only hope that .NET has no hidden features that could provide backdoor functionality.

We don't know for certain that there is a security flaw in Microsoft's DCOM source code that resulted in the vulnerability described in MS03-026 (Q823980) and exploited by the MS Blaster worm. The RPC/DCOM backdoor might have been caused by intentional modifications to the binary after compilation. If the flaw existed in the compiled software product but did not exist in its underlying source code, then a software patch is inadequate to restore trust. It's essential to disclose the cause of the flaw. Mending broken trust is only possible with answers to questions that we wouldn't normally bother to ask.

If a source-code cause can be found in a situation such as MS03-026, proper incident response would be to presume the backdoor was planted on purpose and investigate for evidence that would support this hypothesis. More than any other type of security vulnerability in software, one that results in a backdoor should be analyzed with the view that the defect may have been purposeful. **w::d**