Window Contents Capturing using WM_PRINT Message

Version 1.00: November 4, 2000
Version 1.01: November 6, 2000 (add Within_WM_PRINT method)
Version 1.02: December 1, 2000 (add sample program link)
Version 1.03: April 12, 2004 (add DLL injection to capture window of other processes)

Problem

The normal way of capturing the contents of a window into a bitmap is creating a memory device context (**CreateCompatibleDC**), creating a device-dependent bitmap (**CreateCompatibleBitmap**) or a DIB section (**CreateDIBSection**), selecting the bitmap into the memory DC (**SelectObject**), and then bitblt from the window's device context (**GetWindowDC**) into the memory DC (**Bitblt**). After that, a copy of the contents of the window as it appears on the screen is stored in the bitmap.

But what if the window is hidden, or partially blocked with other windows ? When a window is hidden or partially blocked, the non-visible part of the window will be clipped in the device context returned from GetWindowDC. In other words, that part of the window can't be captured using a simple **BitBlt**.

To capture any window, completely visible, partially visible, or complete invisible, Win32 API provides two special messages, WM_PRINT and WM_PRINTCLIENT. Both these messages take a device context handle as a parameter, and the window handling these messages is supposed to draw itself or its client area into the device context provided.

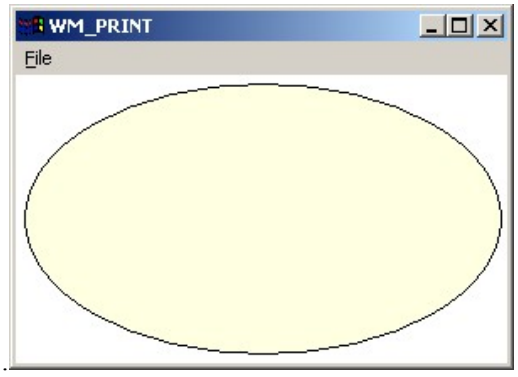
Sounds good ? There is a catch. Normally only windows implemented by the operating system are knowledgeable to handle these messages. If you send a WM_PRINT message to a window, normally all the non-client area, which includes border, title bar, menu bar, scroll bar, etc., and common controls are drawn properly. Client area of windows implemented by application programs are normally left blank.

This article shows a sophisticated method to trick a window implemented by applications to handle WM_PRINTCLIENT message *without* its source code.

Test Program

To experiment with WM_PRINT/WM_PRINTCLIENT messages, a text program is written. Actually, the program is a slightly modified version of the "Hello, World" program generated by MSVC Wizard (plain Win32 application).

The routine handling WM_PAINT message is OnPaint, which calls BeginPaint, a custom drawing routine OnDraw, and then EndPaint. The OnDraw routine just draws an ellipse in the client area. Here is how its screen looks.



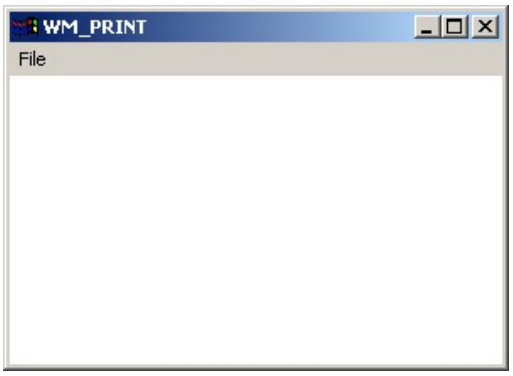

```
void OnDraw(HWND hWnd, HDC hDC)
{
    RECT rt;
    GetClientRect(hWnd, &rt);

    SelectObject(hDC, GetSysColorBrush(COLOR_INFOBK));
    Ellipse(hDC, rt.left+5, rt.top+5, rt.right-3, rt.bottom-5);
}
```

```
void OnPaint(HWND hWnd)
{
    PAINTSTRUCT ps;

    HDC hDC = BeginPaint(hWnd, & ps);

    OnDraw(hWnd, hDC);
    EndPaint(hWnd, & ps);
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_COMMAND:
            switch ( LOWORD(wParam) )
            {
                case IDM_EXIT:
                    DestroyWindow(hWnd);
                    break;

                case ID_FILE_PRINT:
                    PrintWindow(hWnd);
                    break;

                default:
                   return DefWindowProc(hWnd, message, wParam, lParam);
            }
            break;

        case WM_PAINT:
            OnPaint(hWnd);
            break;

        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
   }

   return 0;
}
```

To test the WM_PRINT/WM_PRINTCLIENT message, a "Print" menu item is added to the "File" menu. The window procedure calls the following PrintWindow routine to capture the screen using WM_PRINT message. Note the WM_PRINT message creates a memory DC, a DDB, selects the DDB into the memory DC, and then passes the memory DC handle as the WPARAM of the WM_PRINT message. The LPARAM parameter of the message specifies that everything should be drawn, including client/non-client area, background, and any child window.

```
void PrintWindow(HWND hWnd)
{
    HDC hDCMem = CreateCompatibleDC(NULL);

    RECT rect;

    GetWindowRect(hWnd, & rect);

    HBITMAP hBmp = NULL;

    {
        HDC hDC = GetDC(hWnd);
        hBmp = CreateCompatibleBitmap(hDC, rect.right - rect.left, rect.bottom - rect.top);
        ReleaseDC(hWnd, hDC);
    }

    HGDIOBJ hOld = SelectObject(hDCMem, hBmp);
    SendMessage(hWnd, WM_PRINT, (WPARAM) hDCMem, PRF_CHILDREN | PRF_CLIENT | PRF_ERASEBKGND | PRF_NONCLIENT | PRF_OWNED);

    SelectObject(hDCMem, hOld);
    DeleteObject(hDCMem);

    OpenClipboard(hWnd);

    EmptyClipboard();
    SetClipboardData(CF_BITMAP, hBmp);
    CloseClipboard();
}
```

When WM_PRINT message returns, the bitmap is pasted to the clipboard, so you can use any graphics application to view/save the image. Here is what's being captured, everything except the ellipse.



Prototype Solution

The problem with the test program shown above is of course the disconnection between the WM_PRINT message, and the OnPaint routine handling WM_PAINT message. The default window procedure is smart enough to draw non-client area and then send a WM_PRINTCLIENT message to the window. But there is no default processing for the WM_PRINTCLIENT message, which explains why only the client are is left blank.

If you have the source code of the window procedure, adding a handling of WM_PRINTCLIENT to share the WM_PAINT message handling is very easy, as is shown below.

```
void OnPaint(HWND hWnd, WPARAM wParam)
{
    PAINTSTRUCT ps;
    HDC         hDC;

    if ( wParam==0 )
        hDC = BeginPaint(hWnd, & ps);
    else
        hDC = (HDC) wParam;

    OnDraw(hWnd, hDC);

    if ( wParam==0 )
        EndPaint(hWnd, & ps);
}

LRESULT CALLBACK WndProc0(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        ...
        case WM_PRINTCLIENT:
            SendMessage(hWnd, WM_PAINT, wParam, lParam);
            break;

        case WM_PAINT:
            OnPaint(hWnd, wParam);
        break;
        ...
   }
   return 0;
}
```

In the code shown above, handling for the WM_PRINTCLIENT is added, which just sends a WM_PAINT message to the window itself with the wParam and lParam. The WPARAM parameter is added to OnPaint routine. When wParam is not 0, it's cast into a device context handle, instead of calling BeginPaint to retrive a device context for the non-client area. Likewise, EndPaint is only called when wParam is 0. These simple changes in source code level makes the whole WM_PRINT message handling complete for client area.

Handling WM_PRINTCLIENT Message without Source Code Change

What if you do not have the source code of the window ? Subclassing the window to add a handling for the WM_PRINTCLIENT is easy. What's hard is how to trick bypass BeginPaint and EndPaint, and how to pass the wParam from WM_PRINTCLIENT to the drawing

code after BeginPaint.

Here is the declaraction of the CPaintHook class which handles window subclassing and implementation of WM_PRINTCLIENT message handling.

```
// Copyright (C) 2000 by Feng Yuan (www.fengyuan.com)

class CPaintHook
{
    BYTE      m_thunk[9];
    WNDPROC   m_OldWndProc;
    HDC       m_hDC;

    static HDC  WINAPI MyBeginPaint(HWND hWnd, LPPAINTSTRUCT lpPaint);
    static BOOL WINAPI MyEndPaint(HWND hWnd, LPPAINTSTRUCT lpPaint);

    virtual LRESULT WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

public:

    bool Within_WM_PRINT(void) const
    {
        if ( (m_thunk[0]==0xB9) && ((* (unsigned *) (m_thunk+5))==0x20FF018B) )
            return m_hDC !=0;
        else
            return false;
    }

    void SubClass(HWND hWnd);
};
```

The CPaintHook class has three member variables, one BYTE array of storing some machine code, a pointer to the original window procedure, and a device context handle. The two static methods replaces the original system provided BeginPaint and EndPaint routines. A virtual window message procedure is provided to override message processing for the window. Finally, the SubClass method subclasses an existing window and makes sure it handles WM_PRINTCLIENT message properly.

The implementation of this seemly simple class is quite tricky. Some knowledge of Win32 API implementation, compiler code generation, Intel machine code, and virtual memory is needed to understand it fully.

```
// Copyright (C) 2000 by Feng Yuan (www.fengyuan.com)

#include "stdafx.h"
#include <assert.h>

#include "hookpaint.h"

bool Hook(const TCHAR * module, const TCHAR * proc, unsigned & syscall_id, BYTE * & pProc, const void * pNewProc)
{
    HINSTANCE hMod = GetModuleHandle(module);

    pProc = (BYTE *) GetProcAddress(hMod, proc);

    if ( pProc[0] == 0xB8 )
    {
        syscall_id = * (unsigned *) (pProc + 1);

        DWORD flOldProtect;

        VirtualProtect(pProc, 5, PAGE_EXECUTE_READWRITE, & flOldProtect);

        pProc[0] = 0xE9;
        * (unsigned *) (pProc+1) = (unsigned)pNewProc - (unsigned) (pProc+5);

        pProc += 5;

        return true;
    }
    else
        return false;
}
```

The Hook routine hooks certain kind of exported function from a module by directly modifying its starting machine code. The benefit of hacking machine code directly is that you only need to hack into a single place, all the call in a process is taken care of. But hacking machine code directly is very tricky because it's not easy to parse machine code to find extra space for a five byte jump instruction. Chapter 4 of my book contains more generic code to handle this problem. What's shown here only applies to a special case, which applies to BeginPaint and EndPaint on Windows NT/2000 machines. On these machine, BeginPaint and EndPaint calls system services provided by Win32K.SYS. These routines follow a strict pattern, the first instruction stores a DWORD index into the EAX register. The instructions after that issue a

software interruption (0x2E), which will be served by Win32K.SYS in kernel mode address space.

The Hook routine uses GetModuleHandle to retrieve module handle, GetProcAddress to retrieve the address of an exported Win32 API function. It then checks if the first instruction is a constant move to EAX register instruction (0xB8). If a match is found, VirtualProtect is used to change the protection flag for that page to PAGE_EXECUTE_READWRITE, which makes it writeable. The system service call index is saved, and then the first five bytes are changed to a jump instruction to a function whose address is passed through the pNewProc parameter.

```
LRESULT CPaintHook::WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    assert(m_OldWndProc);

    if ( uMsg==WM_PRINTCLIENT )
    {
        m_hDC = (HDC) wParam;
        uMsg  = WM_PAINT;
    }

    LRESULT hRslt = CallWindowProc(m_OldWndProc, hWnd, uMsg, wParam, lParam);

    m_hDC = NULL;

    return hRslt;
}
```

Implementing CPaintHook::WndProc is fairly simple. If the current message is WM_PRINTCLIENT, the device context handle passed in WPARAM is saved in member variable m_hDC, and then message is changed to WM_PAINT. CallWindowProc is used to call the original window procedure.

```
HDC WINAPI CPaintHook::MyBeginPaint(HWND hWnd, LPPAINTSTRUCT lpPaint)
{
    const CPaintHook * pThis = (CPaintHook *) GetWindowLong(hWnd, GWL_WNDPROC);

    pThis = (const CPaintHook *) ( (unsigned) pThis - (unsigned) & pThis->m_thunk[0] + (unsigned) pThis );

    if ( pThis->Within_WM_PRINT() )
    {
        memset(lpPaint, 0, sizeof(PAINTSTRUCT));

        lpPaint->hdc = pThis->m_hDC;

        GetClientRect(hWnd, & lpPaint->rcPaint);

        return pThis->m_hDC;
    }
    else
    {
        __asm   mov     eax, syscall_BeginPaint
        __asm   push    lpPaint
        __asm   push    hWnd
        __asm   call    pBeginPaint
    }
}

BOOL WINAPI CPaintHook::MyEndPaint(HWND hWnd, LPPAINTSTRUCT lpPaint)
{
    const CPaintHook * pThis = (CPaintHook *) GetWindowLong(hWnd, GWL_WNDPROC);

    pThis = (const CPaintHook *) ( (unsigned) pThis - (unsigned) & pThis->m_thunk[0] + (unsigned) pThis );

    if ( pThis->Within_WM_PRINT() )
        return TRUE;
    else
    {
        __asm   mov     eax, syscall_EndPaint
        __asm   push    lpPaint
        __asm   push    hWnd
        __asm   call    pEndPaint
    }
}
```

Implementation of the two static functions, MyBeginPaint and MyEndPaint, are very similar. Beging static member functions, they do not have 'this' pointer to access object member variables. The two functions calculates the current 'this' pointer from the current window procedure address, which is the address of its m_thunk member variable (explained below). Once 'this' pointer is got, the m_hDC member variable is changed to see if we're actually handling a WM_PRINTCLIENT message, instead of normal WM_PAINT message. If a device context handle is given, the original BeginPaint and EndPaint will be skipped. Otherwise, the system service index is set into the EAX register, and the instructions after the first instruction in the original BeginPaint/EndPaint is called as a subroutine, although a jump instruction

without pusing the parameters will work too.

```
static unsigned syscall_BeginPaint = 0;
static BYTE *   pBeginPaint        = NULL;

static unsigned syscall_EndPaint   = 0;
static BYTE *   pEndPaint          = NULL;

CPaintHook::CPaintHook()
{
    static bool s_hooked = false;

    if ( ! s_hooked )
    {
        Hook("USER32.DLL", "BeginPaint", syscall_BeginPaint, pBeginPaint, MyBeginPaint);
        Hook("USER32.DLL", "EndPaint",   syscall_EndPaint,   pEndPaint,   MyEndPaint);

        s_hooked = true;
    }

    m_thunk[0]              = 0xB9;          // mov ecx,
    *((DWORD *)(m_thunk+1)) = (DWORD) this;  //          this
    *((DWORD *)(m_thunk+5)) = 0x20FF018B;    // mov eax, [ecx]

    m_OldWndProc = NULL;
    m_hDC        = NULL;
}

void CPaintHook::SubClass(HWND hWnd)
{
    m_OldWndProc = (WNDPROC) GetWindowLong(hWnd, GWL_WNDPROC);
    SetWindowLong(hWnd, GWL_WNDPROC, (LONG) ((void *) m_thunk));
}
```

The constructor CPaintHook::CPaintHook and the SubClass method are the magic glue which hold everything together. The constructor will make sure the Hook function are called twice to hook Win32 API function BeginPaint and EndPaint, which are both exported from module USER32.DLL. For each instrance of the CPaintHook class, it's m_thunk data member will be initialized to two machine instructions. The first moves 'this' pointer to the ECX register, the second calls the first virtual method of that object, the CPaintHook::WndProc virtual method implementation.

The SubClass method remembers the original window procedure, and passes the address of m_thunk data member as the new window procedure.

With the CPaintHook class, hooking a window to handle WM_PRINTCLIENT message is a piece of cake. Here is the WinMain function of our test program, which creates an instance of the CPaintHook class on the stack, and calls the SubClass method.

```
int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
    WNDCLASSEX wcex;

    memset(&wcex, 0, sizeof(wcex));

    wcex.cbSize         = sizeof(WNDCLASSEX);
    wcex.style          = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc    = (WNDPROC) WndProc;
    wcex.hInstance      = hInstance;
    wcex.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
    wcex.lpszMenuName   = (LPCSTR) IDC_PAINT;
    wcex.lpszClassName  = "Class";

    RegisterClassEx(&wcex);

    HWND hWnd = CreateWindow("Class", "WM_PRINT", WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

    assert(hWnd);

    CPaintHook hook;

    hook.SubClass(hWnd);

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    MSG msg;
```

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

    return msg.wParam;
}
```

Sample Program

Sample program using WIN32 API wmprint.zip

Sample program for capturing windows of other processes using DLL injection: capture.zip

Limitation

- The Hook function only handles exported function whose first instruction is "MOV EAX, <DWORD_constant>". So the implementation shown here only applies to Windows NT/2000. Refer to Chapter 4 for more generic of restrictive API hooking solutions.
- Only tested on Windows 2000 machine.