Exploiting design flaws in the Win32 API for privilege escalation. Or... Shatter Attacks - How to break Windows

Foon

August 7, 2002

{ivegotta@tombom.co.uk}

1 Introduction

This paper presents a new generation of attacks against Microsoft Windows, and possibly other message-based windowing systems. The flaws presented in this paper are, at the time of writing, un-fixable. The only reliable solution to these attacks requires functionality that is not present in Windows, as well as efforts on the part of every single Windows software vendor. Microsoft has known about these flaws for some time; when I alerted them to this attack, their response was that they do not class it as a flaw - the email can be found here. This research was sparked by comments made by Microsoft VP Jim Allchin who stated, under oath, that there were flaws in Windows so great that they would threaten national security if the Windows source code were to be disclosed. He mentioned Message Queueing, and immediately regretted it. However, given the quantity of research currently taking place around the world after Mr Allchin's comments, it is about time the white hat community saw what is actually possible.

This paper is a step-by-step walkthrough of how to exploit one example of this class of flaw. Several other attack methods are discussed, although examples are not given. There are many ways to exploit these flaws, and many variations on each of the stages presented. This is just one example.

2 Background - the Win32 messaging system

Applications within Windows are entirely controlled through the use of messages. When a key is pressed, a message is sent to the current active window which states that a key was pressed. When Windows decides that an application needs to redraw its client area, it send a message to the application. In fact, when any event takes place that an application needs to know about, it is sent a message. These messages are placed into a queue, and are processed in order by the application.
This is a very reliable mechanism for controlling applications. However, on Win32 the mechanism for controlling these messages is flawed. Any application on a given desktop can send a message to any window on the same desktop, regardless of whether or not that window is owned by the sending application, and regardless of whether the target application wants to receive those messages. There is no mechanism for authenticating the source of a message; a message sent from a malicious application is indistinguishable from a message sent by the Windows kernel. It is this lack of authentication that we will be exploiting, taking into consideration that these messages can be used to manipulate windows and the processes that own them.

3 Overview

In this example, I will be exploiting Network Associates VirusScan v4.5.1, running on Windows 2000 Professional. Since the VirusScan Console runs on my desktop as LocalSystem and I am logged on as a guest user, the objective is to trick VirusScan into running my code to elevate my privileges. This is accomplished in several easy stages.

1. Locate a suitable window within VirusScan (an edit box is perfect), and obtain a window handle to it.

2. Remove any length restrictions that may be present on that edit box, so that I can type in an arbitrary quantity of data.

3. Paste in some binary executable code.

4. Force VirusScan to execute my code (as LocalSystem)

This is actually very easy to do. Windows conveniently provides all of the functionality that we will be needing. I have written a small application called Shatter which implements this functionality. You'll also need a hex editor that is capable of copying binary data to the clipboard $IuseUltraEdit$, and a debugger $IuseWinDbg$.

Windows messages consist of three parts, a message identifier and two parameters. The parameters are used differently depending on what message is sent. This makes our life simpler, since we only have to worry about four things; a window handle to receive the message, the message, and two parameters. Let's find out how easy this is...

4 Stage 1: Locating a window

We need to locate an edit control of some kind - something that we can type stuff into. Don't worry if it's restricted, we can cure that. Fire up the VirusScan console, and hit the first button - "New Task". Conveniently, at the top of the dialog, there's an edit box. That will do perfectly. Now, we need a handle to that control so that we can interact with it. Windows is more than happy to give us a handle to any window we like - we just have to ask it. Fire up Shatter, and position it so that you can still see the VirusScan edit control underneath it. Click on "Get cursor window" - Shatter should add an item in the list box beneath like "102f2 - Get cursor window". This is because we've asked Windows to give us a handle to the window directly underneath the cursor. Move the cursor over the VirusScan edit control and hit Space to trigger Shatter again. Shatter should clear the list box, and tell you the handle for the target window - in my case it's 30270. So, we can now interact programmatically with a window that is running with higher privileges than we are. Let's paste in some shellcode.

5 Stage 2: Removing Restrictions

Now that we have a window handle, we can send any messages we like to that control and it will blindly execute them. First things first - let's make sure we have enough space for our shellcode.

Within Shatter, type your window handle into the "Handle" box. The message to set the maximum text length of an edit box is EM_SETLIMITTEXT. The first parameter is the new maximum text length, and the second parameter is ignored. Type 4 into the WPARAM box, and 0 into the third. Click on EM_SETLIMITTEXT to send the message, and try to type something into the VirusScan edit box. You shouldn't be able to type more than 4 characters. Change the 4 to FFFFFFFF and send the message again. Now try typing into the VirusScan edit box; you now have over 4Gb (theoretically) of space within that edit control. Should be enough for even the most wasteful shellcode.

6 Stage 3: Injecting Shellcode

Next up, let's try pasting something into the box. Yes, OK, you could just right-click and choose Paste, but for the sake of argument let's work as if we couldn't do that. Clear the VirusScan edit box, and fire up Notepad. Type some text into Notepad, and copy it. Back in Shatter, we want to send VirusScan a "Paste clipboard contents" message, which is WM_PASTE. Both parameters for this message should be zero, so set the WPARAM and LPARAM to zero, leaving the handle the same. Click WM_PASTE, and watch your text appear in the VirusScan edit box. Click it again, and it should now be there twice. Fun, huh?

OK, that's enough playing. Clear the VirusScan edit box again, and fire up your hex editor. Load up sploit.bin, included in the Shatter zipfile. This is the shellcode taken from Jill (Hey, Dark Spyrit!) which fires a remote command shell back to you. It's hard-coded to send a command shell to the loopback adress on port 123, so now's probably a good time to fire up a Netcat listener before you forget. Fire up a cmd, hit "nc -lp 123" and forget it. Back to our hex edit. Copy the shellcode to the clipboard, making sure you get all of it (including the FOON at the beginning - we'll need that in a sec). Back to Shatter, and hit the WM_PASTE button again. You should now see a whole load of nasty-looking characters in the VirusScan edit box; that's our shellcode, nicely pasted in.

7 Stage 4: Executing the code

This is the only part of the process that requires any skill. Fire up your debugger, and attach it to the avconsol.exe process (Using WinDbg, that's F6 to attach, and just choose the process). Next, do a search through memory for the FOON string. The WinDbg command is **s -a 00000001 10000000 "FOON"** but you might use a different debugger. Note down the memory location that the string appears at; it'll probably appear a couple of times, don't ask me why. Any of them will do. On my system, the shellcode appears at 0x00148c28, it shouldn't be far off if you're using the same version. Now, kill the debugger, log on as a guest user, and prepare to receive localsystem privs. Follow stages 1 through 3 again, noting that everything still works as a guest user. Don't forget the Netcat listener to receive the shell.

At this point, you might be thinking that attaching a debugger is a privileged operation. It is. However, much the same as when writing a buffer overflow exploit, you can do that part on any system; all you need is the load address which should then work on any system running the same version of the software. In actual fact, you needn't actually do this at all. Most applications have their own exception handlers (VirusScan certainly does), so if they generate an access violation, they just deal with it and move on rather than crashing. So, there's nothing to stop you pasting in a few hundred kilobytes of NOPs and then just iterating through memory until you finally hit the right address and your shellode executes. Not particularly elegant, but it'll work.

The final message that we're going to make use of is WM_TIMER. This is a slightly odd and very dangerous message, since it can contain (as the second parameter) the address of a timer callback function. If this second parameter is non-zero, execution will jump to the location it specifies. Yes, you read that right; you can send any window a WM_TIMER message with a non-zero second parameter (the first is a timer ID) and execution jumps to that address. As far as I know, the message doesn't even go into the message queue, so the application doesn't even have the chance to ignore it. Silly, silly, silly...

So, within Shatter, the handle should be set to the VirusScan edit control

containing our shellcode. The first parameter can be anything you like, and the second parameter should be 512 bytes or so above the address we picked out of the debugger earlier (we have 1K of NOP's in front of the shellcode, so we should land slap bang in the middle of them); on my system that's 0x148c28 + 0x200 = 0x148e28. Hit WM_TIMER, and your netcat listener should come alive with a command prompt. A quick WHOAMI will reveal that you have indeed gone from guest to local system. Enjoy.

8 Alternative techniques

There's a few other ways of doing what we just managed, utilising the same basic mechanisms but maybe adding a bit more complexity. The EM_GETLINE message tells an edit control to copy its contents to a location specified within the message. How would you like to write arbitrary quantities of data to arbitrary locations in memory? How easy a sploit do you want? We've seen how the restrictions can be removed from the length of an edit control; what happens when an application depends on these restrictions? When an application expects 16 bytes of data from a limited-to-16-byte edit box, we can type in a few gigs. Everyone, on three; 1....2....3....Buffer Overflow! Probably stack-based too, since 16 bytes of data is unlikely to come from the heap. Also, when we send WM_TIMER, the parameter we specify as a timer ID gets pushed onto the stack along with a whole load of other crap. It's not inconceivable that we could find a function which makes use of the 3rd function parameter and none of the others, allowing us to jump directly to a sploit with a single message.

Talking of the heap, that's another great thing about these exploits. Generally, applications will create dialog boxes on the heap well in advance of any major memory operations taking place; our shellcode address is going to remain pretty static. In my experience it rarely moves more than 20 bytes between instances. Static jump addresses shouldn't be a problem, but who cares? Send the app an EM_GETLINE message so it writes your shellcode to a location you specify (Hell, overwrite the heap. Who's gonna care?) and then specify the same address in your WM_TIMER message. A completely NOP-free sploit! What fun!

9 Fixing the problem

Okay, so this is pretty easy to exploit. How is everyone gonna fix this? I can see two quick and dirty methods which will break a whole lotta functionality, and one very long-winded solution which is never going to be a total solution. Let me explain.

1. Don't allow people to enumerate windows Nasty. Multiple breakages. Theoretically possible, but I'd hate to see people trying to work around not knowing what windows are on the desktop when they need to.

2. Don't allow messages to pass between applications with different privileges Means that you couldn't interact with any window on your desktop that's not running as you; means that VirusScan at the very least (probably most personal firewalls, too) would need a whole lotta redesigning.

3. Add source info to messages, and depend on applications to decide whether or not to process the messages Would need an extension to the Win32 API, and a whole lotta work for people to use it. Big job, and people would still get it wrong. Look at buffer overflows - they've been around for years, and they're still fairly common.

Basically, there is no simple solution, which is why Microsoft have been keeping this under their hat. Problem is, if I can find this, I can guarantee that other people have as well. They might not tell anyone about it, and the next time they get into your system as a low-priv user, you wouldn't have a clue how they got LocalSystem out of it. After all, you're all up to date on patches, aren't you?

10 Addendum: Why is this a problem?

When Microsoft saw a copy of this paper, they sent me a response stating clearly that they are aware of these attacks, and they do not class them as vulnera-

bilities. I believe that this point of view is incorrect. The two reasons that Microsoft stated are that a) They require unrestricted physical access to your computer, or b) they require you to run some kind of malicious code on your machine. I agree completely that in both of these scenarios, 0wning the machine is pretty easy. However, they've missed the point. These are techniques that an attacker can use to escalate their privileges. If they can get guest-level access to a machine, these attacks allow you to get localsystem privileges from any user account. Anyone ever heard of a little tool called hk.exe? How about ERunAsX (AKA DebPloit)? How about iishack.dll? All of these tools exploit some flaw that allows you to escalate your privileges AFTER you've gained access to the machine. All of these have been recognised as security holes by Microsoft, and patched.

If you have a corporate desktop machine, most commonly those machines will be quite tightly locked down. The user on that machine cannot do very much that they have not been explicitly granted permission to do. If that machine is vulnerable to a shatter attack, that user can gain localsystem privileges and do what they like. Even worse is the case of Terminal Services (or Citrix). Imagine a company providing terminal service functionality to their clients, for whatever purpose. That company is NOT going to give their users any real privileges. Shatter attacks will allow those users to completely take over that server; localsystem privileges are higher than the Administrator, and on a shared server that's a problem. Oh, and it doesn't require console access either - I've successfully executed these attacks against a Terminal Server a hundred miles away.

The simple fact is that Microsoft KNOW that they cannot fix these flaws. The mechanism used is the Win32 API, which has been fairly static since Windows NT 3.5 was released in July 1993. Microsoft cannot change it. The only way they could stop these attacks is to prevent applications from running on the desktop with privileges higher than those of the user logged on. Microsoft believe that the desktop is a security boundary, and that any window on it should be classed as untrusted. This is true, but only for Windows, and because of these flaws. Either way, Microsoft break their own rules; there's numerous windows on a standard desktop that run as localsystem. Use my shatter tool to verify this - there's a whole load of unnamed windows which might be running as

Localsystem, and a few invisible windows (like the DDE server) that definitely are. Security boundary my arse.

11 Is this just a Win32 problem?

Probably, yes. The only mainstream competitor to Windows in terms of windowing systems is X windows. X is based on a similar underlying technique, that of queueing messages that are passed between windows. X, however, has two major differences. Firstly, a window in X is just a window - it's a blank page on which the application can do what it likes. Unlike Win32 where each control is a window in its own right, a control in X is just a picture. When you click that control, you're actually clicking the window surrounding it, and the application is responsible for figuring out whether or not there's actually a control underneath your mouse and responding accordingly. Secondly, and more importantly, X messages are just notifications, not control messages. You can't tell an X window to do something just by sending it a message. You can't tell it to paste text. You can't tell it to change the input limits on a control. You certainly can't tell it to jump to a location in memory and start executing it. The best you can do is send it the mouse clicks or keyboard strokes that correspond to a paste command - you certainly can't tell a control to paste in the contents of the clipboard. As such, it's still theoretically possible for some of these attacks to work against X but in practice it's highly unlikely. You could flood an application with fake messages and see how it responds; you could send it corrupt messages and see how it responds. Chances are, it would cope just fine, since it'll choose what to do with the messages and process the flood one at a time.

Anyway kids, have fun, play nicely, be good. And remember - if it ain't broke, hit it again.