Why .shared sections are a security hole

devblogs.microsoft.com/oldnewthing/20040804-00

August 4, 2004

Raymond Chen

Many people will recommend using shared data sections as a way to share data between multiple instances of an application. This sounds like a great idea, but in fact it's a security hole.

Proper shared memory objects created by the CreateFileMapping function can be secured. They have security descriptors that let you specify which users are allowed to have what level of access. By contrast, anybody who loads your EXE or DLL gets access to your shared memory section.

Allow me to demonstrate with an intentionally insecure program.

Take the scratch program and make the following changes:

```
#pragma comment(linker, "/SECTION:.shared,RWS")
#pragma data_seg(".shared")
int g_iShared = 0;
#pragma data_seg()


void CALLBACK TimerProc(HWND hwnd, UINT, UINT_PTR, DWORD)
{
  int iNew = g_iShared + 1;
  if (iNew == 10) iNew = 0;
  g_iShared = iNew;
  InvalidateRect(hwnd, NULL, TRUE);
}


BOOL
OnCreate(HWND hwnd, LPCREATESTRUCT lpcs)
{
    SetTimer(hwnd, 1, 1000, TimerProc);
    return TRUE;
}


void
PaintContent(HWND hwnd, PAINTSTRUCT *pps)
{
  TCHAR sz[2];
  wsprintf(sz, TEXT("%d"), g_iShared);
  TextOut(pps->hdc, 0, 0, sz, 1);
}
```

Go ahead and run this program. It counts from 0 to 9 over and over again. Since the TimerProc function never lets g_iShared go above 9, the wsprintf is safe from buffer overflow.

Or is it?

Run this program. Then use the runas utility to run a second copy of this program under a different user. For extra fun, make one of the users an administrator and another a non-administrator.

Notice that the counter counts up at double speed. That's to be expected since the counter is shared.

Okay, now close one of the copies and relaunch it under a debugger. (It's more fun if you let the administrator's copy run free and run the non-administrator's copy run under a debugger.) Let both programs run, then break into the debugger and change the value of the variable g_iShared to something really big, say, 1000000.

Now, depending on how intrusive your debugger is, you might or might not see the crash. Some debuggers are "helpful" and "unshare" shared memory sections when you change their values from the debugger. Helpful for debugging (maybe), bad for my demonstration (definitely).

Here's how I did it with the built-in ntsd debugger. I opened a command prompt, which runs as myself (and I am not an administrator). I then used the runas utility to run the scratch program as administrator. It is the administrator's copy of the scratch program that I'm going to cause to crash even though I am just a boring normal non-administrative user.

From the normal command prompt, I typed "ntsd scratch" to run the scratch program under the debugger. From the debugger prompt, I typed "u TimerProc" to disassemble the TimerProc function, looking for

```
01001143 a300300001       mov     [scratch!g_iShared (01003000)],eax
```

(note: your numbers may differ). I then typed "g 1001143" to instruct the debugger to execute normally until that instruction is reached. When the debugger broke, I typed "r eax=12341234;t" to change the value of the eax register to 0x12341324 and then trace one instruction. That one-instruction trace wrote the out-of-range value into shared memory, and one second later, the administrator version of the program crashed with a buffer overflow. What happened?

Since the memory is shared, all running copies of the scratch program have access to it. ALl I did was use the debugger to run a copy of the scratch program and change the value of the shared memory variable. Since the variable is shared, the value also changes in the administrator's copy of the program, which then causes the wsprintf buffer to overflow, thereby crashing the administrator's copy of the program.

A denial of service is bad enough, but you can really do fun things if a program keeps anything of value in shared memory. If there is a pointer, you can corrupt the pointer. If there is a string, you can remove the null terminator and cause it to become "impossibly" long, resulting in a potential buffer overflow if somebody copies it without checking the length.

And if there is a C++ object with a vtable, then you have just hit the mother lode! What you do is redirect the vtable to a bogus vtable (which you construct in the shared memory section), and put a function pointer entry in that vtable that points into some code that you generated (also into the shared memory section) that takes over the machine. (If NX is enabled, then the attack is much harder but still possible in principle.)

Even if you can't trigger a buffer overflow by messing with variables in shared memory, you can still cause the program to behave erratically. Just scribbling random numbers all over the shared memory section will certainly induce "interesting" behavior in the program under

attack.

Moral of the story: Avoid shared memory sections. Since you can't attach an ACL to the section, anybody who can load your EXE or DLL can modify your variables and cause havoc in another instance of the program that is running at a higher security level.

Raymond Chen

Follow