

Buffer Overflows

Mark Russinovich
(From Mark Russinovich Blog)

Buffer Overflows

No, I'm not talking about the kind of buffer overflows that viruses can take advantage of to inject malicious code onto other systems, I'm talking about the kind that, if you use Filemon or Regmon, you've probably seen in their traces. If you've never noticed one, fire up one of those two tools and after collecting a log of system-wide activity, find an example by searching for "buffer overflow". Here's an example of file system buffer overflow errors:

csrss.exe:972	OPEN	C:\WINDOWS\WinSxS\manifests\amd64...	SUCCESS	Options: Open Sequential Access: All
csrss.exe:972	READ	C:\WINDOWS\WinSxS\manifests\amd64...	SUCCESS	Offset: 0 Length: 2
csrss.exe:972	CLOSE	C:\WINDOWS\WinSxS\manifests\amd64...	SUCCESS	
csrss.exe:972	OPEN	C:\WINDOWS\WinSxS\manifests\amd64...	SUCCESS	Options: Open Sequential Access: All
csrss.exe:972	QUERY INF...	C:\WINDOWS\WinSxS\manifests\amd64...	BUFFER OVERFLOW	FileFsVolumeInformation
csrss.exe:972	QUERY INF...	C:\WINDOWS\WinSxS\manifests\amd64...	BUFFER OVERFLOW	FileAllInformation
csrss.exe:972	READ	C:\WINDOWS\WinSxS\manifests\amd64...	SUCCESS	Offset: 0 Length: 4095
csrss.exe:972	READ	C:\WINDOWS\WinSxS\manifests\amd64...	END OF FILE	Offset: 1864 Length: 8178
csrss.exe:972	CLOSE	C:\WINDOWS\WinSxS\manifests\amd64...	SUCCESS	

Do these errors indicate a problem? No, they are a standard way for the system to indicate that there's more information available than can fit into a requester's output buffer. In other words, the system is telling the caller that if it was to copy all the data requested, it would overflow the buffer. Thus, the error really means that a buffer overflow was avoided, not that one occurred.

Given that a buffer overflow means that a requester didn't receive all the data that they asked for you'd expect programmers to avoid them, or when they can't, to follow with another request specifying a buffer large enough for the data. However, in the Filemon trace neither case applies. Instead, there are two different requests in a row, each resulting in buffer overflow errors. In the first request the Csrss.exe process, which is the Windows environment subsystem process, queries information about a file system volume and in the second request it queries information about a particular file. It doesn't follow up with successful requests, but continues with other activity.

The answer to why Csrss.exe doesn't care that its requests result in errors lies in the type of requests it's making. A program that queries volume information using Windows APIs is underneath using the NtQueryVolumeInformationFile API that's exported by Ntdll.dll, the Native API export DLL (you can read more about the Native API here). There are several different classes of information that a program can query. The one that Csrss is asking for in the trace is FileFsVolumeInformation. The Windows Installable File System (IFS) Kit documents that for that class a caller should expect output data to be formatted as a FILE_FS_VOLUME_INFORMATION structure, which looks like this:

```
typedef struct _FILE_FS_VOLUME_INFORMATION {
    LARGE_INTEGER VolumeCreationTime;
    ULONG VolumeSerialNumber;
    ULONG VolumeLabelLength;
    BOOLEAN SupportsObjects;
    WCHAR VolumeLabel[1];
} FILE_FS_VOLUME_INFORMATION, *PFILE_FS_VOLUME_INFORMATION;
```

Notice that the first four fields in the structure have a fixed length while the last field, VolumeLabel, has a size that depends on the length of the volume's label string.

When a file system driver gets this type of query it fills in as much information as fits in the caller's buffer and, if the buffer is too small to hold the entire structure, returns a buffer overflow error and the size of the buffer required to hold all the data. I suspect that Csrss is really only interested in the

Buffer Overflows

Mark Russinovich
(From Mark Russinovich Blog)

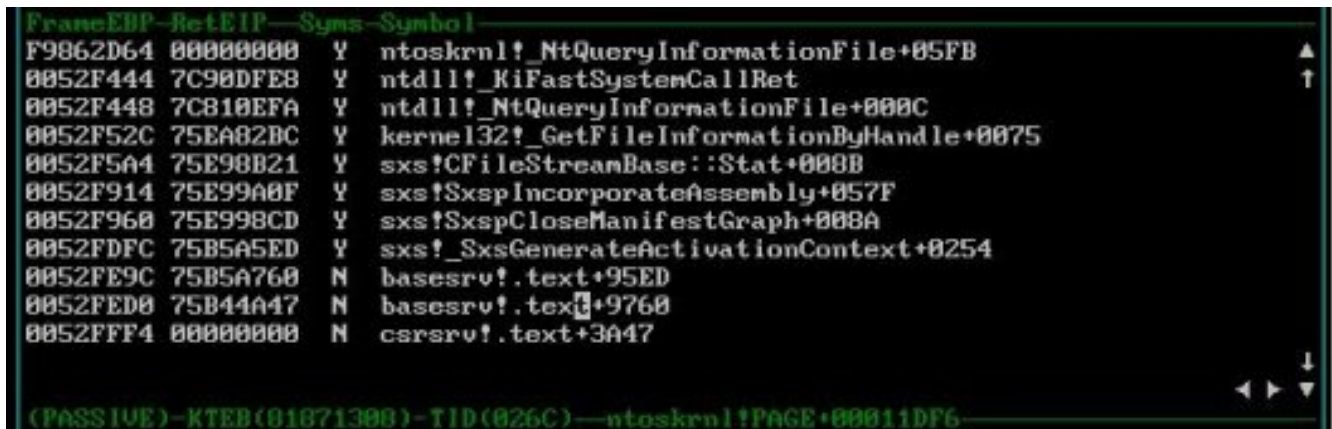
volume creation time and therefore passing in a buffer only large enough to hold the first part of the structure. The file system driver fills that part in, and because the volume label won't fit in Csrss's buffer, returns an error. However, Csrss has gotten the information it wanted and ignores the error.

The second buffer overflow has a similar explanation. Csrss is querying information about a file using the FileAllInformation class of NtQueryInformationFile. The IFS Kit documents the output structure as:

```
typedef struct _FILE_ALL_INFORMATION {
FILE_BASIC_INFORMATION BasicInformation;
FILE_STANDARD_INFORMATION StandardInformation;
FILE_INTERNAL_INFORMATION InternalInformation;
FILE_EA_INFORMATION EaInformation;
FILE_ACCESS_INFORMATION AccessInformation;
FILE_POSITION_INFORMATION PositionInformation;
FILE_MODE_INFORMATION ModeInformation;
FILE_ALIGNMENT_INFORMATION AlignmentInformation;
FILE_NAME_INFORMATION NameInformation;
} FILE_ALL_INFORMATION, *PFILE_ALL_INFORMATION;
```

Again, the only variable length field is the last one, which stores the name of the file being queried. If Csrss doesn't care about the name, only the information preceding it in the structure, it can pass a buffer only large enough to hold those fields and ignore the buffer overflow error.

Incidentally, a stack trace of the second buffer overflow reveals this:



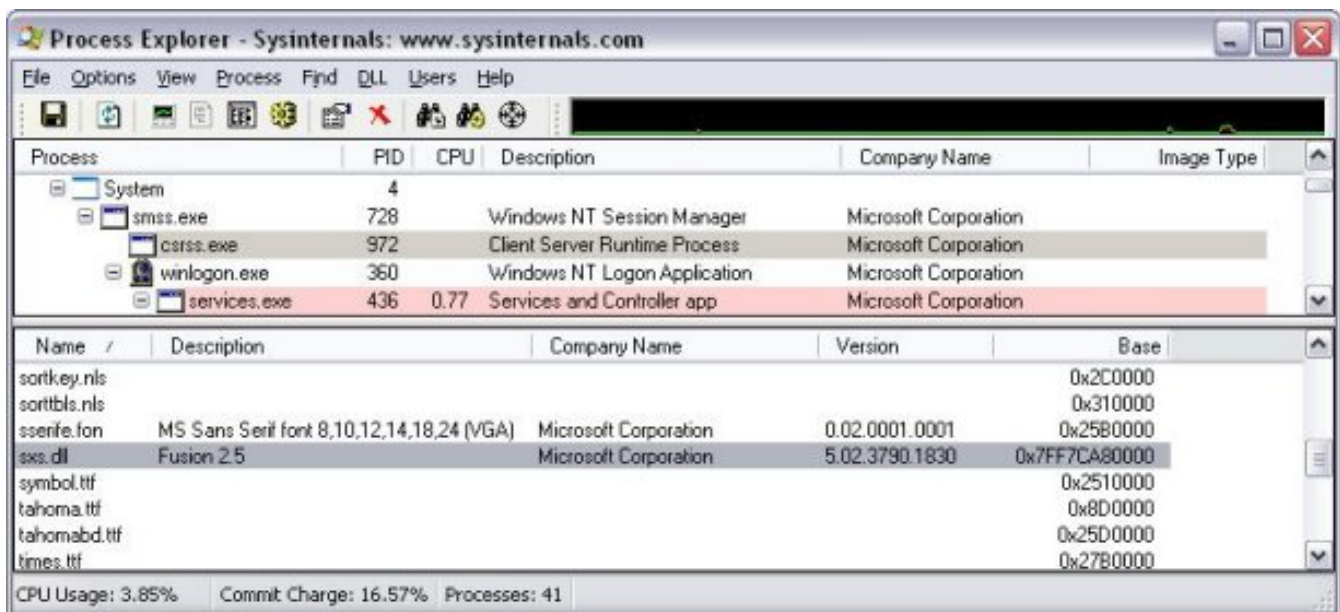
```
FrameEBP--RetEIP--Syms--Symbol
F9862D64 00000000 Y ntoskrnl!_NtQueryInformationFile+05FB
0052F444 7C98DFEB Y ntdll!_KiFastSystemCallRet
0052F448 7CB18EFA Y ntdll!_NtQueryInformationFile+000C
0052F52C 75EA82BC Y kernel32!_GetFileInformationByHandle+0075
0052F5A4 75E98B21 Y sxs!CFileStreamBase::Stat+000B
0052F914 75E99A0F Y sxs!SxspIncorporateAssembly+057F
0052F968 75E998CD Y sxs!SxspCloseManifestGraph+008A
0052FDFC 75B5A5ED Y sxs!_SxsGenerateActivationContext+0254
0052FE9C 75B5A768 N basesrv!.text+95ED
0052FED8 75B44A47 N basesrv!.text+9768
0052FFF4 00000000 N csrssrv!.text+3A47

(PASSIVE)-KTEB(81871300)-TID(826C)-ntoskrnl!PAGE+00011DF6
```

What is the "sxs" module? A look at the sxs DLL in Process Explorer's DLL View of the Csrss process shows this:

Buffer Overflows

Mark Russinovich
(From Mark Russinovich Blog)



SxS is the "Fusion" DLL, which a little research will show manages the Side-by-Side Assembly storage that allows multiple versions of the same DLLs to exist in harmony on a system. SxS is calling `GetFileInformationByHandle`, which is a Windows API documented in the Platform SDK. The API takes a file handle as input and returns a buffer formatted as a `BY_HANDLE_FILE_INFORMATION` structure:

```
typedef struct _BY_HANDLE_FILE_INFORMATION {
    DWORD dwFileAttributes;
    FILETIME ftCreationTime;
    FILETIME ftLastAccessTime;
    FILETIME ftLastWriteTime;
    DWORD dwVolumeSerialNumber;
    DWORD nFileSizeHigh;
    DWORD nFileSizeLow;
    DWORD nNumberOfLinks;
    DWORD nFileIndexHigh;
    DWORD nFileIndexLow;
} BY_HANDLE_FILE_INFORMATION;
```

All of the information returned in this structure, except for the volume serial number, is also returned in the `FILE_ALL_INFORMATION` structure. You can therefore probably guess where the call to `NtQueryVolumeInformationFile` that occurs immediately prior to the `NtQueryInformationFile` call originates: `GetFileInformationByHandle` first queries the volume in order to get its serial number.

Our investigation shows that the buffer overflow errors seen in the Filemon trace are errors expected by the `GetFileInformationByHandle` API, which is simply avoiding the need to allocate buffers large enough to hold information it's not interested in. The bottom line is that buffer overflow errors in a Filemon trace are not an indication that there's a security problem and are usually not due to bad

Buffer Overflows

Mark Russinovich
(From Mark Russinovich Blog)

programming.

Buffer Overflows in Regmon Traces

Last time I talked about buffer overflow errors that you might see in Filemon traces. Now I'll turn my attention to the same errors, but in Regmon traces. Recall that a buffer overflow error in this context is not a security hole, but a way for the system to tell an application that there's more data available in response to a query the application has made than can fit in the application's output buffer. A commenter on the previous post pointed out that a better error for this case would be "buffer too small" or "more data available", and I agree. There is in fact a STATUS_BUFFER_TOO_SMALL error code, but it's used in situations where no data is copied to the caller's buffer, whereas STATUS_BUFFER_OVERFLOW is used when some, but not all, available data has been copied.

Buffer overflow errors in Regmon traces are relatively common. The documentation for RegQueryValueKey says:

If the buffer specified by lpData parameter is not large enough to hold the data, the function returns ERROR_MORE_DATA and stores the required buffer size in the variable pointed to by lpcbData. In this case, the contents of the lpData buffer are undefined.

If lpData is NULL, and lpcbData is non-NULL, the function returns ERROR_SUCCESS and stores the size of the data, in bytes, in the variable pointed to by lpcbData. This enables an application to determine the best way to allocate a buffer for the value's data.

There are two programming approaches commonly taken to reading variable-length Registry data. The first is to simply pass in a NULL pointer on the first call and if the call succeeds to allocate a buffer for a second query. The other approach is for an application to first try and use a static buffer of a size that the programmer expects will usually be large enough to receive the stored data. If the system reports ERROR_MORE_DATA, which is the Windows API equivalent of the native STATUS_BUFFER_OVERFLOW error, the application dynamically allocates a buffer of the required size and re-executes the query. The second approach has the advantage of avoiding a second query in most cases.

Given these approaches you should never see two buffer overflow errors in a row that query the same data. At least that's what you'd expect, but on many occasions I've seen two buffer overflow errors followed by a third query of the same data that succeeds. Here's an example I see on a Windows XP SP2 system that results when I open My Computer:

explorer.exe:244	CloseKey	HKLM\Software\Microsoft\Internet Explorer\Toolbar	SUCCESS	
explorer.exe:244	CreateKey	HKCU\Software\Microsoft\Internet Explorer\Toolbar	SUCCESS	Access: 0x2001F
explorer.exe:244	OpenKey	HKCU\Software\Microsoft\Internet Explorer\Toolbar\ShellBrowser	SUCCESS	Access: 0x20019
explorer.exe:244	QueryValue	HKCU\Software\Microsoft\Internet Explorer\Toolbar\ShellBrowser\NTBarLayout	BUFFER_OVERFLOW	
explorer.exe:244	QueryValue	HKCU\Software\Microsoft\Internet Explorer\Toolbar\ShellBrowser\NTBarLayout	BUFFER_OVERFLOW	
explorer.exe:244	QueryValue	HKCU\Software\Microsoft\Internet Explorer\Toolbar\ShellBrowser\NTBarLayout	SUCCESS	11 00 00 00 4C 00 00 ...
explorer.exe:244	CloseKey	HKCU\Software\Microsoft\Internet Explorer\Toolbar\ShellBrowser	SUCCESS	

I decided to investigate this particular case as I was writing this blog posting. First, I started SoftICE and set a breakpoint on NtQueryValueKey. When the breakpoint triggered I single-stepped in SoftICE to the return instruction, at which point the status of the query is stored in the EAX register. I then cleared the first breakpoint, and set a conditional breakpoint on the return instruction for the case that the result is STATUS_BUFFER_OVERFLOW (0x80000005):

```
bpx _NtQueryValueKey+0346 IF eax==80000005
```

Buffer Overflows

Mark Russinovich
(From Mark Russinovich Blog)

Then I opened My Computer and this is the stack at first buffer overflow breakpoint:

```
Frame EBP-Ret EIP-Sym-Symbol
F98D24A4 00000000 Y ntoskrnl!_NtQueryValueKey+0346
018EF78C 7C90E20A Y ntdll!_KiFastSystemCallRet
018EE790 77DD6EDC Y ntdll!_NtQueryValueKey+000C
018EE878 77DD7930 Y advapi32!_LocalBaseRegQueryValue+0111
018EE8D0 77F7B0F9 Y advapi32!_RegQueryValueExA+00AD
018EE8FC 77F7A359 Y shlwapi!_SHOpenRegStream2A+0091
018EEB38 760021F2 Y shlwapi!_SHOpenRegStream2W+00AB
018EED68 75FB460A Y browseui!_AorW_OpenRegStream+0074
018EEF98 75FB4642 Y browseui!GetRegStream+006F
018EEFAC 75FA46E7 Y browseui!GetITBarStream+0015
018EEFBC 75FABD4B Y browseui!CShellBrowser2::_GetITBarStream+0016
018EEFE4 75FAC233 Y browseui!CShellBrowser2::_PrepareInternetToolbar+00BA
018EF04C 7777CDEA Y browseui!CShellBrowser2::OnCreate+03F7
018EF068 75FA405B Y shdocvw!_CBaseBrowser2_CreateInstance+004A
018EF084 75FAFC93 Y browseui!CCommonBrowser::_WndProcBS+0020
018EF0C0 75FAD668 Y browseui!CShellBrowser2::_WndProcBS+0196
018EF0EC 77D48734 Y browseui!IEFrameWndProc+00FF
018EF118 77D4D05B Y user32!_UserCallWinProcCheckWow
018EF180 77D4B4C0 Y user32!_GetAsyncKeyState+000A
018EF1D4 77D4FD29 Y user32!_NtUserBeginPaint+000F
018EF204 7C90EAE3 Y user32!_HMValidateHandleNoSecure+0022
018EF73C 77D501F7 Y ntdll!_KiUserCallbackDispatcher+0013
018EF7E8 77D4FF83 Y user32!_SetClassLongA+001C
018EF824 77F7A7B1 Y user32!_SendMessageTimeoutA+0062
(PASSIVE) - KTEB(D197B020) - TID(04E4) - ntoskrnl!PAGE+00009C00
```

A look at the parameters Shlwapi passes into RegQueryValueExA revealed a NULL buffer pointer. I stepped back out into Shlwapi's code and saw that if RegQueryValueExA returns ERROR_SUCCESS, which is the error code to which RegQueryValueExA translates STATUS_BUFFER_OVERFLOW if the caller uses a NULL buffer pointer, Shlwapi dynamically allocates a buffer of the required size and makes the call again:

```
SHOpenRegStream2A+0091
001B:77F7B0E9 PUSH EBX
001B:77F7B0EA LEA ECX,[EBP+14]
001B:77F7B0ED PUSH ECX
001B:77F7B0EE PUSH EBX
001B:77F7B0EF PUSH DWORD PTR [EBP+10]
001B:77F7B0F2 PUSH EAX
001B:77F7B0F3 CALL [__imp__RegQueryValueExA]
001B:77F7B0F9 TEST EAX,EAX
001B:77F7B0FB JNZ 477F7B131
001B:77F7B0FD CMP [EBP+08],EBX
001B:77F7B100 JZ 477F7B131
001B:77F7B102 PUSH DWORD PTR [EBP+08]
001B:77F7B105 MOV ECX,EDI
001B:77F7B107 CALL CMemStream::GrowBuffer
001B:77F7B10C TEST EAX,EAX
001B:77F7B10E JZ 177F7A373
001B:77F7B114 LEA EAX,[EBP+08]
001B:77F7B117 PUSH EAX
001B:77F7B118 PUSH DWORD PTR [EDI+08]
001B:77F7B11B LEA EAX,[EBP+14]
001B:77F7B11E PUSH EAX
001B:77F7B11F PUSH EBX
001B:77F7B120 PUSH DWORD PTR [EBP+10]
001B:77F7B123 PUSH DWORD PTR [ESI]
001B:77F7B125 CALL [__imp__RegQueryValueExA]
001B:77F7B12B MOV EAX,[EBP+08]
001B:77F7B12E MOV [EDI+10],EAX
001B:77F7B131 PUSH 12
001B:77F7B133 CALL _IsOS
001B:77F7B138 TEST EAX,EAX
```

Buffer Overflows

Mark Russinovich
(From Mark Russinovich Blog)

Shlwapi makes one call into RegQueryValueExA, but two entries get added to Regmon's trace, one with a buffer overflow and one that's successful. Stepping into Advapi32's implementation of RegQueryValueExA showed me why the second call results in two Registry queries in the Regmon trace. NtQueryValueKey returns a data structure storing several different pieces of information regarding the Registry value, including its type, length and title index:

```
typedef struct _KEY_VALUE_PARTIAL_INFORMATION {
    ULONG TitleIndex;
    ULONG Type;
    ULONG DataLength;
    UCHAR Data[1]; // Variable size
} KEY_VALUE_PARTIAL_INFORMATION, *PKEY_VALUE_PARTIAL_INFORMATION;
```

However, RegQueryValueExA returns only the actual data into the caller's output buffer. RegQueryValueExA (and RegQueryValueExW) therefore uses a 144-byte stack-allocated buffer to query the value. This is the place in Advapi32 where RegQueryValueExA makes that call:

```
001B:77DD6EB7 MOV     ESI,[__imp__NtQueryValueKey]
001B:77DD6EBD LEA    EAX,[EBP-00A4]
001B:77DD6EC3 PUSH   EAX
001B:77DD6EC4 MOV    EDI,00000090 ; "K."
001B:77DD6EC9 PUSH   EDI
001B:77DD6ECA LEA    EAX,[EBP-0094]
001B:77DD6ED0 PUSH   EAX
001B:77DD6ED1 PUSH   02
001B:77DD6ED3 PUSH   EBX
001B:77DD6ED4 PUSH   DWORD PTR [EBP-00B0]
001B:77DD6EDA CALL   ESI
```

The first instruction in the disassembly is RegQueryValueExA loading the address of NtQueryValueKey into the ESI register. 144 is represented as 0x90 in hexadecimal, which you can see loaded into the EDI register and then passed as the fifth parameter to NtQueryValueKey, which corresponds to the output buffer size.

The additional value information included in the KEY_VALUE_PARTIAL_INFORMATION structure requires 12 bytes, so if the data being read is larger than 132 bytes NtQueryValueKey returns STATUS_BUFFER_OVERFLOW. If the caller's buffer is large enough to store the data RegQueryValueExA allocates a buffer large enough to hold it and then executes the call to NtQueryValueKey again. After getting the data RegQueryValueExA copies it to the caller's buffer.

When I saw the double-buffer overflows in the Regmon trace I thought it was evidence of poorly-written code, but my investigation showed that the pattern will be present in an application that reads Registry values larger than 132 bytes in size. What I don't know is why the Windows API developers picked 132-bytes as the magical buffer size they expected would hold most Registry data.