VIRUS ANALYSIS

Unexpected Resutls [*sic*]

Peter Ferrie
Symantec Security Response, Australia

In early 2000, while studying the latest release of the Portable Executable format documentation from *Microsoft*, I noticed the word 'callback' in a section describing data initialization. The section was called 'Thread Local Storage (TLS)'; in previous revisions of the documentation I had disregarded it, considering it uninteresting, but this time it had my full attention.

Where there are callbacks, there is executable code and where there is executable code, there may be viruses. However, it was a further two years before the appearance, in 2002, of the first virus that is aware of Thread Local Storage: W32/Chiton.

Blast from the Past

The virus writer's handle will be familiar to some. Calling himself 'roy g biv', perhaps from the colours of the rainbow, in 1993 he was the author of a virus that used the circular partition trick to make it difficult to boot from a floppy disk (by exploiting a bug that exists in MS-DOS v4.00-7.00 – see *VB*, September 1995, p.12).

It seems that, once again, roy g biv has created a piece of malware that may cause a few headaches for the anti-virus developers.

Modern History

All threads of a process share the address space and global variables of that process.

For applications that use a fixed number of unique threads, each thread can allocate memory and store the pointer in a separate global variable that the programmer has reserved for the purpose.

A problem exists for applications whose maximum thread count cannot be determined, or is unreasonably large, or those which execute multiple instances of a single thread. In these cases, there is no easy way to reserve a unique memory location for each of the threads, without the use of Thread Local Storage.

Thread Local Storage is a special storage class that is supported by *Windows NT*, *Windows 2000* and *Windows XP*. There are two types: dynamic and static.

Dynamic Thread Local Storage

Dynamic Thread Local Storage is used by applications containing threads whose size of local data is not constant, or which could not be determined when the application was compiled.

An application uses dynamic Thread Local Storage in the following way.

When a process is created, it allocates a Thread Local Storage index by calling the TLSAlloc() API. This index is stored in a global variable of the process.

Each thread that is created will allocate memory for its local data, then pass a pointer to this memory, and the process' Thread Local Storage index, to the TLSSetValue() API. At any time, a thread can retrieve the pointer to its local data by passing the process' Thread Local Storage index to the TLSGetValue() API.

The TLSGetValue() and TLSSetValue() functions access a table in the operating system's memory which is updated dynamically whenever a thread-switch occurs. This is how a single index can be used by all threads to access individual values.

Static Thread Local Storage

Static Thread Local Storage is used by applications in which all threads use a data block of the same size and contents (initially, at least). These data are described by a Thread Local Storage template.

In addition to the template, an array of callbacks can exist to provide customized initialization for each thread. Each of these callbacks is called before the process begins executing (the DLL_PROCESS_ATTACH event) and after the process stops executing (the DLL_PROCESS_DETACH event).

The callbacks are also called before a thread begins executing (the DLL_THREAD_ATTACH event) and after a thread stops executing (the DLL_THREAD_DETACH event), unless the DisableThreadLibraryCalls() API has been called first. In that case, only the process events will cause the callbacks to be called.

Since the callbacks are accessed via an array that is pointed to by a table, the address of which is stored in the tenth data directory in the Portable Executable header, this could be considered by anti-virus software (and researchers) to be an entry-point obscuring method (if they are not aware of Thread Local Storage data).

It is easy to understand why the virus author calls this technique 'the hidden entry point'.

Round and Round

The first time W32/Chiton is executed, it checks the event that caused its execution. The virus replicates only during the DLL_PROCESS_DETACH event, which occurs when an application is exiting. The reason for this is that the virus uses the Structured Exception Handler list to gain access to KERNEL32.DLL. This file is not present in the list at an earlier time.

After gaining access to KERNEL32.DLL, the virus will retrieve the addresses of API functions that it requires, using the increasingly common CRC method to match the names.

Unlike the authors of some of the other viruses that use the CRC method, the author of W32/Chiton was aware that APIs must be stored in alphabetical order, so there is no need to search the CRC table repeatedly.

Additionally, the virus has support for both ANSI and Unicode functions merged into a single routine, and selects the set of APIs that is appropriate to the current platform (ANSI for *Windows 9x* and *ME*; Unicode for *Windows NT*, *2000* and *XP*).

The virus searches for files in the current directory and all subdirectories, using a linked-list instead of a recursive function. This is important from the point of view of the virus author, because the virus infects DLLs, whose stack size can be very small.

Filters

Files are examined for their potential to be infected, regardless of their suffix, and will be infected if they pass a very strict set of filters.

The first of these filters is the support for the System File Checker that exists in *Windows 98/ME/2000/XP*. The virus author was aware of the fact that the IsFileProtected() API requires a Unicode path, while directory searching on *Windows 9x* and *ME* require an ANSI path, so the virus transforms the path dynamically.

The remaining filters include the condition that the file being examined must be a character mode or GUI application for the Intel 386+ CPU, that the file must have no digital certificates, and that it must have no bytes outside of the image.

Touch and Go

When a file is found that meets the infection criteria, it will be infected. If relocation data exist at the end of the file, the virus will move the data to a larger offset in the file, and place its code in the gap that has been created. If there are no relocation data at the end of the file, the virus code will be placed here.

The infection will then proceed in one of two ways, depending on the file type.

For DLLs, the Thread Local Storage method is not used because a DLL will not call the TLS callbacks if the DLL is loaded using the LoadLibrary() API (and perhaps the virus author was concerned about the virus being labelled a WinNT virus, rather than a Win32 virus). Instead, the entry-point is altered to point directly to the virus code.

However, for EXE files, the Thread Local Storage method is used. The virus carries its own Thread Local Storage directory, which will be used should the target file contain no directory at all. The virus carries its own callback array for those hosts whose Thread Local Storage directory contains no callbacks.

When it encounters a host that already has a Thread Local Storage directory containing callbacks, the virus will save the address of the first callback and replace it with the address of the virus code.

Once the infection is complete, the virus will calculate a new file checksum, if one existed previously, before continuing to search for more files.

Once the file searching has finished, the virus will allow the application to exit by forcing an exception to occur. This technique appears a number of times in the virus code, and is an elegant way to reduce the code size, in addition to functioning as an effective anti-debugging method.

Since the virus has protected itself against errors by installing a Structured Exception Handler, the simulation of an error condition results in the execution of a common block of code to exit a routine. This avoids the need for separate handlers for successful and unsuccessful code completion.

Conclusion

It seems that some old dogs can learn new tricks. The author of W32/Chiton has moved successfully from the DOS platform to the Win32 platform, found a feature in the *Windows* Portable Executable file format that had (until now) been overlooked by anti-virus developers, and found a way to exploit it.

Additionally, the virus author distributed a document along with the virus source, which describes some further infection methods using Thread Local Storage. Interesting times lie ahead.

W32/Chiton

Aliases:	W32/Shrug.
Type:	Direct-action parasitic appender/ inserter.
Infects:	*Windows* Portable Executable files.
Payload:	None.
Removal:	Delete infected files and restore them from backup.