VIRUS ANALYSIS

HIDAN AND DANGEROUS

Peter Ferrie
Symantec Security Response, USA

One of the things that almost all anti-malware researchers have in common is a copy of *Interactive DisAssembler* (*IDA*). It is perhaps the best tool we have for disassembling files, since it is capable of so many important things: it displays the file more or less as it really appears in memory, applying relocations, and resolving imports. *IDA* can follow all of the code paths and note all of the data references, comment the API parameters, and even determine the stack parameters.

Since some people have custom requirements, *IDA* also supports a plug-in interface. Plug-ins can do many things and control many of *IDA*'s actions – including directing it to infect files.

Enter the latest member of the ever-growing W32/Chiton family. The author of the virus calls this one 'W32/Hidan'.

WHERE ARE YOU HIDING?

When Hidan is started for the first time, it queries the default value of the 'HKCR\.idb' registry key. The virus author assumes that if this registry key is present, then *IDA* must also be present on the system. The registry key is created when the *Windows* GUI version of *IDA* is started for the first time. However, the command-line version of *IDA* does not create this key. What's more, there are other tools, such as *Microsoft Visual Studio*, that use the same registry key, so its presence on the system is no guarantee that *IDA* is present.

Regardless of which application created the registry key, the registry value will contain the name of the handler. In the case of *IDA*, the value is 'IDApro.Database32'. The virus then queries the default value of its 'shell\open\command' subkey. The data usually contain the following string:

```
"<path>\idag.exe" "%1"
```

The virus searches this string for the second quote, while remembering the location of the last backslash. Once the second quote is found, the virus appends the string 'plugins' after the last backslash, so the string becomes '<path>\plugins'. This is the directory in which *IDA* keeps its plug-ins.

BEND AND STRETCH

The virus decompresses and drops a plug-in file in the plugins directory, using the (fixed) filename 'hidan.plw'. The file contains only the virus code.

As with the other viruses in the Chiton family, this one is aware of the techniques that are used against viruses that drop files, and will work around all of the commonly used countermeasures: if a file exists already, its read-only attribute (if any) will be removed, and the file will be deleted. If a directory exists instead, then it will be renamed to a random name.

The structure of the dropped file is similar to that of the W32/OU812 member of the W32/Chiton family. However, Hidan differs in one way: the entrypoint of the file is inside the file header itself, which can complicate analysis slightly (particularly when using *IDA*). The file is also not constant, because the virus knows which bytes in the file header are not used, and replaces them with a value chosen randomly at the time of dropping the file.

After dropping the file, the virus runs the host code.

PLUGGING THE HOLE

Apart from dropping the file, the virus performs no other actions in infected files. It simply waits until *IDA* loads the viral plug-in.

When *IDA* starts, it loads all plug-ins that correspond to the platform on which it is running. *IDA* runs on the 32-bit and 64-bit versions of *Windows*, and the 32-bit and 64-bit versions of *Linux*. Additionally, in the *Professional* version of *IDA*, there is a 32-bit *Windows* version that supports 64-bit addressing, so it is capable of loading 64-bit files on a 32-bit machine.

Each of these versions has its own plug-in format and suffix to distinguish them. The suffix 'plw' means that the plug-in is for the 32-bit *Windows* version of *IDA*; 'x86' means that the plug-in is for the 64-bit *Windows* version; 'p64' means that the plug-in is for the 32-bit *Windows* version of *IDA* that supports 64-bit addressing. The suffix 'plx' means that the plug-in is for the 32-bit *Linux* version of *IDA*, and the suffix 'plx64' means that the plug-in is for the 64-bit *Linux* version of *IDA*.

IDA SYMBOLISM

Internally, plug-ins are ordinary DLLs on the *Windows* platform, or shared libraries on the *Linux* platform. Plug-ins must export one special symbol, called either 'PLUGIN' or '_PLUGIN', or it must be ordinal 1 on the *Windows* platform.

The symbol is a pointer to a plug-in structure. The structure contains a field which holds the version number of the SDK used to produce the plug-in. That version number must match the version that *IDA* is expecting, otherwise it will

refuse to load the plug-in. There are two versions of the virus: one is compatible with *IDA 4.8*, and one is compatible with *IDA* version 4.9. While the SDK version did not change between *IDA* versions 4.9 and 5.0 (the current version at the time of writing), the behaviour of *IDA* did.

THE TERMINATOR

The plug-in structure contains three pointers to functions, 'init', 'term', and 'run'. The init function is used to initialize the plug-in. The term function is used to terminate the plug-in. The run function is used to run the main part of the plug-in.

In *IDA* prior to version 5.0, if a plug-in init function returned a value of 0, which means that the plug-in cannot or did not want to load (perhaps because the environment is not compatible, or the file to examine is not of the correct format, etc.), *IDA* would free the plug-in directly (via the FreeLibrary() API on the *Windows* platform). Thus, neither the run nor term functions would be called, so the virus author did not include them in the virus.

However, in *IDA* version 5.0, if the term function pointer is non-zero, then *IDA* will call the term function before calling FreeLibrary(). Since the virus does not support this function (there are data at that location, but not a function pointer), the virus crashes at that point, taking *IDA* down with it. This seems a silly bug just to save four bytes of zeroes.

Unfortunately, the virus has already done its work by the time it crashes, since the init function contains the replication code.

INITIATION CEREMONY

When the viral plug-in init function is called, it begins by retrieving some file infection-related API addresses from kernel32.dll, the IsFileProtected() API address from sfc.dll (or sfc_os.dll if the platform is *Windows XP/2003*), and two undocumented symbols from ida.wll (RootNode and netnode_value in the .A version, or netnode_valstr() in the .B version).

When the netnode API is called with the value held by the RootNode symbol, an ANSI-format pathname is returned. That pathname corresponds to the file being examined by *IDA*. The virus converts that pathname to Unicode format, then passes it to the IsFileProtected() API.

If the file is not protected, then it will be infected only if it passes a very strict set of filters. These 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

If the file 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 own 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 entrypoint is altered to point to the virus code. The virus will calculate a new file checksum, if one existed previously.

Once the infection is complete, the virus forces an exception to occur in order to terminate the replication code and return to the init function. The init function then returns a value of zero to *IDA*, to signal that the plug-in should be unloaded.

The interesting thing is that since *IDA* has already started to load the now-infected file, the change to the entrypoint is not visible – though the virus code can be seen if one knows where to look for it, and if the file is reloaded, the full changes are visible.

Of course, it would be possible for a plug-in to interfere with all of that – the plug-in could remain in memory and intercept disk accesses. It could also restore the host entrypoint label and remove the viral one. However, this seems like an even more pointless exercise than writing the virus in the first place.

CONCLUSION

This member of the Chiton family is just a proof-of-concept virus for a new platform, created by a virus author who specialises in them. Given its simplicity, Hidan might be considered to be hiding in plain sight.

W32/Chiton variant

Type:	Memory-resident parasitic appender/inserter.
Size:	1,321 bytes (.A), 1,330 bytes (.B)
Infects:	*Windows* Portable Executable files.
Payload:	None.
Removal:	Delete infected files and restore them from backup.