MALWARE ANALYSIS 1

PROPHET AND LOSS

Peter Ferrie
Microsoft, USA

The release of the long-delayed EOF-rRlf-DoomRiderz virus zine probably marks the last of its kind. While the quality is not terribly high, there are some viruses of interest. A series of analyses in alphabetical order begins with this one: W32/Divino.

I'M A LOCAL

The virus begins by storing the selector of the local descriptor table in the ImageBase field in the PEB, and then reading four bytes and checking if the result is non-zero. A non-zero result should always occur, because the top half of the ImageBase field will remain untouched and non-zero. This might be an anti-emulator trick for an emulator that stores four bytes instead of two. However, it seems more likely that what the virus author had in mind was to read only two bytes and detect whether the local descriptor table (LDT) is in use, but had to reverse the condition because of the extra bytes that the virus reads. The use of the LDT is a characteristic of virtual machines such as *VMware* and *VirtualPC*, along with *Norman*'s *SandBox*.

In any case, if the result is zero, the virus attempts to continue execution, but without decrypting itself first. When that happens, the virus crashes and the application terminates. If the result is non-zero, then the virus decrypts the first stage of its body and attempts to transfer control to it, using an address that was calculated from values in the PE header at the time of infection. This means that the virus is not aware of 'Address Space Layout Randomization' (ASLR). If the infected file was built to be ASLR-aware, then the virus will crash and the application will terminate.

UN-SafeSEH

The first stage of the virus registers a structured exception handler, then intentionally causes an exception. This is an old anti-debugging trick which any good debugger can skip easily enough. Since the handler appears immediately after the call to the anti-debugging routine, it's a simple matter to step over the call and continue execution. However, the virus is not aware of 'SafeSEH', which overrides the legacy structured exception handling. If the infected file was built with SafeSEH, then the exception that the virus raises will cause the application to exit, because the exception address will not match any known address.

The virus unregisters the handler, copies a decryptor to the stack, and then attempts to execute the decryptor from there. The virus is not aware of 'Data Execution Protection' (DEP). If the infected file was built to be DEP-aware, then the virus will crash and the application will terminate.

BYTE, BYTE BABY

The virus retrieves an address from the stack that points within the kernel32 BaseThreadInitThunk() function. Using this as a starting point, the virus performs a brute-force search in memory for the 'MZ' header. The search is performed byte by byte, rather than on 64KB boundaries, making it slow and inefficient. The virus does not register a structured exception handler for this operation. As a result, the technique fails on *Windows Vista64*. This is because the kernel32.dll in *Windows Vista64* uses a 64KB section alignment, so the region between the file header and the first section is not mapped. Any attempt to access this memory will cause an exception which is not intercepted by the virus. If an exception occurs, the virus will crash and the application will terminate.

ONWARD AND FORWARD

In the event that everything is okay, the virus calculates a pointer 4KB below the current stack pointer value. The virus author assumed that it would remain untouched but this is not always the case, as we will see below. The virus resolves a set of API addresses from kernel32.dll that are required to infect files. The resolver uses checksums instead of names. The list includes GetLastError(), but this is never used. In fact, almost one third of the APIs that are resolved are not used. The reason for not using GetLastError() is clear. It is because on *Windows XP*, the function is forwarded to ntdll.dll, so the address that is exported from kernel32.dll does not point directly to the function. The lack of forwarding support is a common problem for export resolvers in virus code, though the problem is not limited to viruses. There are many runtime packers that also use the checksum technique, which also do not support export forwarding. The reason for not using some of the other APIs is less clear, but the fact that some of them would be used to start a new process and watch its execution suggests that an alternative spreading mechanism was planned but not implemented.

The virus resolves a set of API addresses from ws2_32.dll, some of which will be used to perform a denial-of-service attack as part of the payload. Some of the APIs are not used, but could be used to form the basis for a remote control mechanism, or perhaps an auto-updating capability which may also have been planned but not implemented. The virus also resolves a set of API addresses from shell32.dll and

user32.dll, which will be used to perform some actions on the clipboard.

HANGING BY A THREAD

At this point, the virus copies back the bytes replaced by the first decryptor and creates a thread to run the host code. This allows the virus to achieve per-process residency, however this behaviour can be considered a bug. The problem is that if the host code terminates, the virus code will be forcibly terminated, too. This can result in an interrupted infection and a corrupted file.

The virus retrieves the first four bytes of the GetProcAddress() function and compares it to the 'enter 0,2' instruction. The virus wants to exit if there is a match. This might be an anti-emulator detection. However, since the 'enter' instruction is only three bytes long, the comparison will probably always fail. This behaviour appears to be a bug.

IT'S PAYBACK

The virus carries a payload whose trigger is the execution of an infected file on the 28th day of any month. The date is acquired by allocating a buffer requesting the date format, then comparing the contents to '28'. There is a bug in this routine, however, which is that the buffer is never freed.

The payload exists in two parts. The first part of the payload displays a message box. The message title is the two-digit date, so it is always '28'. The message body is:

```
Win32.Divinorum
Code By Fakedminded/EOF-Project
Mikko cut ur ponytail!
```

The 'Mikko' in the text is presumed to be a reference to *F-Secure*'s Mikko Hyppönen, who happens to wear his hair long.

The second part of the payload attempts to perform a denial-of-service attack on *F-Secure*'s European website. However, now the stack problem appears. The IsValidLocale() function, which is used by the MessageBox() function, uses so much of the stack on *Windows Vista* that it corrupts the API table. The result is that on *Windows Vista* the rest of the payload crashes, and the application terminates.

The virus author appears to be aware of the general problem, since the WSASocket() function also requires a lot of stack, but the virus saves and restores the stack state in order to survive that call.

The denial-of-service attack is effectively limitless, since it uses a 'loop' instruction which relies implicitly on the value in the ecx register. The ecx register is set as a side effect of the call to the Sleep() function. The value is the return address of the call to the EH_epilog() function, which is always greater than 1.

WINNERS AND LOSERS

The virus allocates some memory to hold the current directory name. Another bug exists here, however, which is that the buffer is never freed. The virus retrieves the current directory, and compares the first four bytes with 'WIN' and 'win'. The virus author intended to avoid the '%windir%' directory, which is by default 'WINNT' on *Windows NT* and *Windows 2000*, and 'WINDOWS' for all other platforms. However, there is a bug in the comparison: by comparing four bytes against a three-byte string, the only names that can match are 'WIN' and 'win'.

The virus searches within the current directory for all files with the '.exe' suffix. For each file that is found, the virus opens it and reads the entire file into memory, regardless of how large it is. The virus searches within the entire file for the 'msco' string. This is intended to match 'mscorlib.dll' and similar strings, to avoid the infection of *Microsoft .NET* framework files. There are simpler ways to detect such files, of course, such as the presence of the CLR Runtime Header Data Directory.

Only now does the virus check for the 'MZ' and 'PE' signatures within the file. Another bug exists here, which is that the 'PE' signature comparison is incomplete. The true signature is four bytes long, but the virus checks for only the first two bytes. While it is unlikely that any DOS programs contain such a signature, the possibility exists, and the virus might attempt to infect one as a result.

CHECKS AND BALANCES

The virus performs some simple checks to see if the file can be infected, however these checks are insufficient. The virus checks if the virtual size of the entrypoint section is zero, and if there is sufficient space to add a new section header. The file will not be infected if either of these checks fail. In the case of a section with a virtual size of zero, the physical size should be used instead – perhaps the virus author was not aware of this.

The virus does not check for 64-bit format files. As a result, such files will be infected, but incorrectly. The structure is not damaged, but the virus calculates some absolute addresses using the ImageBase field in the PE header, and in 64-bit files the field is 64 bits large and begins four bytes earlier. The result is that the top four bytes of the ImageBase are referenced instead.

The check for sufficient space for the section header is not quite correct either. The virus author intended to check whether there are 40 individual bytes available, but the virus compares four bytes at a time while advancing one byte at a time. The result is a check for three bytes more than is required.

If there is sufficient space to add a new section header, then the virus appends a new section to the file, and changes the original section names to 'UPXn', where 'n' is an increasing single-digit number. However, the new section is always named 'UPX0'. The virus fills with 'FF' values any remaining space after the new section header. This serves as the infection marker, since now it will appear that there is no space for another section.

REALLY 'NO EXECUTE'

The virus replaces completely the characteristics for the entrypoint section. It changes them to read/write/init, and does the same for the newly added section. This act is not compatible with DEP, since without the Executable flag set in the section header, the contents of the sections cannot be executed on platforms that support DEP.

The new section header states that the section begins immediately after the last section in the file. The virus checks for data that are appended outside of the image, but the check is for the presence of at least 10,000 bytes. Anything smaller than that, such as debug information, will be ignored by the virus. The virus makes some adjustments to the PE header, then writes the entire file back to the disk. At this point, the virus seeks the location of the host entrypoint and writes the first decryptor.

The virus allocates a buffer to hold a copy of the virus body, then copies itself to the buffer and encrypts the copy. Yet another bug exists here, which is that a buffer is allocated for each file to infect, but it is never freed. The result can be a very large allocation of memory if there are a lot of files.

The virus then seeks to the end of the file and writes the encrypted virus body. If there are appended data after the original last section, then they will be 'sandwiched' between the image and the virus body.

The infection is now complete, and the virus searches for another file and repeats the infection process. Once all files have been examined, the virus steps up one directory level and repeats the process, including the stepping, twice. Thus, the virus infects the current directory and two directories above it.

REMOVERS AND SHAKERS

After infecting the nearby files, the virus allocates several buffers in preparation for the final stage. As before, these buffers are never freed. The virus retrieves the list of drive letters. It skips the 'A:' drive, then looks for drive letters that represent removable media. There is a bug in the enumeration, which is that to determine when the end of the list is reached, the virus checks one byte after the current position. The correct behaviour is to check the byte in the current position, because if a debug heap is active, one byte after the current position is the beginning of the 'BAADFOOD' sequence, which continues to the end of the buffer.

If a removable drive is found, then the virus copies itself to the root directory of the drive, as 'driver_setup.exe'. After a short delay, the virus creates an 'autorun.ini' in the root directory of the drive, which contains a reference to the 'driver_setup.exe'.

CLIP GO THE SHEARS

The virus queries the clipboard for objects that are in the process of being copied. For each such object, the virus queries its file attributes. A bug exists here, which is that the virus author did not seem to realize that attributes can be combined. The virus wants to avoid directories and read-only files, but this is only successful if only those attributes are set. However, if a file has read-only and system attributes set, for example, or if a directory is hidden, then the virus will assume that both of those cases describe files.

If a file is found, then the virus will remove the filename from the path to produce a directory. If a directory is found, then the virus will call the infection and stepping routine to infect the nearby files.

The clipboard query returns the number of objects on the clipboard, and the virus is aware of this, but there is some missing code which would be used to enumerate these objects. Instead, only the first object is examined, and a value is left on the stack.

After the virus examines the first object on the clipboard, the virus repeats the dropper and clipboard procedures, beginning with the retrieval of the drive letters. However, a value is left on the stack each time that part of the code is reached, eventually resulting in a stack fault. This causes the virus to crash and the application to be terminated.

CONCLUSION

The virus author wanted to call this virus 'Divinorum', which means 'diviner', someone who can predict something about the future. Here's my prediction: your talents will be recognized and suitably rewarded.

Now read that again.