MALWARE ANALYSIS

WHITHER THE HARUMF?

Peter Ferrie
Microsoft, USA

The second in our series of analyses of viruses contained in the EOF-rRlf-DoomRiderz virus zine is that of W32/Harumf.

'A' 'S'ILLY 'L'ITTLE 'R'EPEAT

The virus begins by decrypting the first stage of its body and attempts to transfer control to it using an address that it calculates 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 has been built to be ASLR-aware, then the virus will crash and the application will terminate. This is not a good way to start.

The decryptor is oligomorphic, having only very few variations, which are taken from a fixed set.

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.

One could be forgiven for thinking that we are looking once again at W32/Divino [1], since the two viruses were written by the same person, and they clearly share some code (and many of the same bugs).

The virus unregisters the handler and then decrypts its second stage. Here is a near bug. The decryptor works only if the virus code is of even length. The reason for this is a combination of instructions and parameters that are not supposed to go together. There is a subtraction, then a comparison, and then a branch. The problem is in several parts. The subtraction is by two, and the branch is taken only if the carry and zero flags are both clear. This would work regardless of the size of the code, if it were not for the comparison. The comparison is made with zero, for which the carry flag can never be set, and the zero flag is set only if the result is zero. The zero flag cannot be set if the virus length is not even. The proper branch instruction would be one that checks the sign flag instead of the carry flag.

This problem is not present in W32/Divino because in that case, the decryptor uses an addition and a comparison of a value that is larger than zero, so the following branch works as intended, regardless of the size of the code.

I'M A LOCAL

The virus stores the selector of the local descriptor table onto the stack, and then reads four bytes and checks if the result is non-zero. A non-zero result should always occur because the location on the stack holds the previous stack frame when the process started, which is always an address above the 64 KB boundary. As a result, the top half of the stack frame 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*.

Next comes a specific detection for *Norman*'s *SandBox*, using a variation of a finding that was described in [2]. In this case, the attack is that *Norman*'s *SandBox* returns the same information for the CPUID instruction, regardless of the index that is specified.

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 64 KB 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 64 KB section alignment, so the region between the file header and the first section are 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.

NEW YEAR'S RESOLUTION

The virus resolves a set of API addresses from kernel32.dll that are required to infect files, using the standard GetProcAddress() method. As a result, the names are clearly visible in the code. Despite this, however, three of the resolved functions are not used.

The virus also resolves a set of API addresses from advapi32.dll that are required to replace a registry value. The RegQueryValueExA() function address is also resolved but not used, because the virus does not care about the previous content of the data that it will replace.

At this point, the virus copies back the bytes replaced by the first decryptor and then begins the search for files to infect.

INFECTIOUS GROOVES

The virus allocates some memory to hold the name of the current directory. There is a bug in this code, however, which is that the memory is never freed. The virus enumerates all objects in the current directory, and looks for anything whose name ends with '.exe'. The virus assumes that such an object is a file. This is a minor bug, but it has no effect here. The reason it has no effect is because the virus attempts to load the object into memory. A directory will cause an error to be returned, which the virus intercepts. This also happens to filter out 64-bit files. How fortunate for the virus author.

CHECKS AND BALANCES

The virus attempts to find a particular resource within the file, whose presence is the infection marker. The resource is a data type with an identifier of 1234. If the resource is not found, then the virus checks within the file for the 'MZ' and 'PE' signatures and the presence of a resource data directory. 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. Of course, the initial load of the file would fail if the file is not in Portable Executable format, so the check for the signatures is redundant.

The virus attempts to copy 512 bytes of data from the host entrypoint, but without checking if there are at least that many bytes available to copy. If the entrypoint is located less than 512 bytes from the end of the image, then the virus will crash and the application will terminate.

The virus walks the section table once to check for pure virtual sections. If any are found, then the virus will skip the file. The virus walks the section table again to check for a section whose name begins with '.rsr'. This is intended to find the '.rsrc' section, but because the full name is not checked, there could be other sections that are matched instead. This could cause trouble later. The virus also requires that this section is the last one in the image.

NOT VERY RESOURCEFUL

If all goes well, the virus unloads the file and then allocates some memory to hold a copy of the virus body. There is a bug in this code, which is that the memory is never freed. The virus encrypts the second stage at this point. Now we reach the 'feature' of the virus. The virus attempts to inject itself as a resource. It uses the resource-updating APIs to do that. However, there is a problem. In order to update a resource, one must specify its language. The virus uses a generic English language selection, which restricts the scope of infection. As a result, any file with multi-language user interface (MUI) resources (the default for many files in *Windows Vista*) will not be infected because the exact language (primary and sub-language) must match.

If the resource updating succeeds, then the virus opens the infected file and requests the file size. The virus allocates some memory to hold a copy of the entire file. There is a bug in this code, which is that the memory is never freed. The virus walks the section table to find the section that contains the entrypoint, and walks the section table yet again to find the section whose name begins with '.rsr' (even though we know that it's the last section – no-one said that this code is optimal). There is another bug here. If the entrypoint is not in any section, then the virus will search beyond the end of the table. It will probably find something that covers the entrypoint value, but the results will be unpredictable.

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 resource 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 virus searches the entire resource section to find the copy of itself. This is certainly simpler than parsing the resource data, but the virus searches only for the first four bytes of its code, which can easily match graphical data and other things. If the virus 'finds' itself, it encrypts the first stage. If the match was in fact false, then the results could be messy.

Finally, the virus copies the decryptor to the host entrypoint and writes the updated data to the file. Another bug exists here, which is that the file handle is never closed. The result is that one handle is leaked for each infected file.

At this point, the virus searches for another object and repeats the process until nothing more can be found.

BUT WAIT, THERE'S MORE

The local replication part of the virus ends here. Then begins the remote replication part. The virus begins by retrieving the process path name, and searches within the last eight characters for the 'haru' string. The significance of the

'haru' string will be described below. Meanwhile, there are two bugs in this code. The first is that the comparison is case-sensitive. This bug is minor, since the virus is likely to have been the creator of the file. The second bug is that the virus does not verify the entire name. This has an effect later.

If the 'haru' string is found, then the virus wants to run 'explorer.exe' with the drive letter of the drive that contains the file. However, there is a bug in this code. The virus constructs the string on the stack, but does so below the current stack pointer, instead of allocating stack space. Then it calls the GlobalAlloc() function to allocate some memory to hold a copy of the string. The problem is that the API call destroys the string. In any case, the allocated length is also off by one byte, which causes heap corruption when the 'string' is copied there. There is also another bug, which is that the memory is never freed.

HARU ICHIBAN!

If somehow the string survived, then a directory listing is displayed for the specified drive. At this point, the virus checks if the payload should run, and also runs two other replication methods, before exiting silently. It is here that the effect of the incomplete 'haru' check appears. The problem is that if an infected file contained the 'haru' string in the name, then the host code will not executed any more. Given that 'Haru' can be a person's name in Japanese (who remembers the author of LHARC?), and it also means 'spring' (the season), there is certainly the possibility of encountering files that contain the string. It may be a rare bug, but it is still a bug.

If the 'haru' string is not found, then the virus performs an additional check before checking if the payload should run, and running two other replication methods. For some reason, one of the methods is executed twice in both cases. Perhaps another method was intended to be included.

The additional check that the virus performs is whether the user is a member of the Administrators group. It uses the IsUserAnAdmin() function, which is documented by *Microsoft* as available in *Windows 2000* and later, but it appears to be present only in *Windows XP* and later. The function is a nice wrapper around code that checks the token membership. If the user is not a member of the Administrators group, then the virus displays the message 'You need Administrator Privilege to run this Application', and then exits. Otherwise, the virus attempts to retrieve the address of the InitializeSRWLock() function. This function was introduced in *Windows Vista*, and its presence or absence provides a method of determining the platform without the use of the GetVersionExA() function (whose results are currently being faked by some anti-malware emulators).

If the virus is running on *Windows Vista*, then the virus sets to zero the 'EnableLUA' value in the 'HKLM\Software\Microsoft\Windows\CurrentVersion\Policies\System' key. However, this has no immediate effect. It requires a reboot of the system for the change to be applied.

REMOTE CONTROL

The payload activates on the 9th of each month. It attempts to download a picture and place it in the root directory of the C: drive. On *Windows Vista*, standard users cannot write to that location, so the download fails in that case. If the download is successful, then the virus waits three seconds before displaying the picture. The picture is a banner that says 'Saddam's Family'. Rather than being a family photo, it's the logo of a heavy metal band which goes by that name.

The first additional replication method is that the virus copies itself as 'vista_crack.exe' to some P2P shared folders, assuming that they exist. The relevant P2P applications are *KaZaA Lite*, *KaZaA*, *EDonkey2000*, *ICQ*, *eMule*, *Gnucleus*, *KMS*, and *LimeWire*.

The second additional replication method begins by getting the bitmap of currently connected drives. For each drive, the virus allocates memory to hold the name of the current directory. There is a bug in this code, which is that the memory is never freed. The virus changes to the root directory of the drive, and then searches for files to infect. The virus also copies itself as 'harulf.exe' to the root directory of the drive, and drops an 'autorun.inf' file that contains a reference to the 'harulf.exe' file. This is the reason for the 'haru'check above. For removable media, the 'autorun.inf' file will run the 'harulf.exe' file when the drive is connected. Since the copied file is also an infected file, the virus does not want the host code to run at that point. The virus checks all 32 bits of the map, even though there can be only 26 drive letters. This might also be considered a bug.

CONCLUSION

Perhaps the funniest thing in this virus, even more than the numerous bugs, is the virus author misspelling his own name: 'coded by fakedmnded!'. Oops, 'i' did it again.

REFERENCES

[1] Ferrie, P. Prophet and Loss. Virus Bulletin, September 2008, p.4. http://www.virusbtn.com/vba/2008/09/vb200809-prophet-loss.

[2] Ferrie, P. Attacks on more virtual machines. http://pferrie.tripod.com/papers/attacks2.pdf.